This document contains suggested enhancements to the Advanced Library Format, using ALF 2.0 as baseline. The document serves as a worksheet rather than a formal proposal. The suggested enhancements are collected in no particular order. The idea is to keep track of evolving proposals here and then agree formally whether or not they should be part of the IEEE spec.

The following template is used throughout this document:

X.0 Item

relation to ALF 2.0 reference to ALF 2.0 chapter

relation to ALF IEEE reference to ALF IEEE chapter

History date of initial draft, date of revisions

X.1 Motivation

Explain reason for new feature

X.2 Proposal

Describe new feature

1.0 Level definition for Vector Expression Language

relation to ALF 2.0 5.3, 5.4, 11.3

relation to ALF IEEE

History initial draft April 16 2001 by Wolfgang
reviewed and rejected by Study Group April 16
rejection confirmed by Tim Ehrler May 1
changed title and closed May 4 by Wolfgang

1.1 Motivation

The vector expression language is a new concept which has almost no equivalent in legacy library model description languages. Currently there are EDA tools which support a subset of the vector expression language. Purpose of this proposal is to re-write the definitions in such a way that it is easy to identify subsets for different levels of support. For example: level0=basic subset, level1=intermediate subset, level2=full set in ALF 2.0, level3=full set in ALF 2.0 plus new proposed extensions.

1.2 Proposal

Level 0: single event, single event & boolean condition, two-event sequence

Level 1: N-event sequence, N-event sequence & boolean condition, alternative event sequence

Level 2: everything in ALF 2.0 (except if we decide to drop something fundamentally unpractical or un-implementable)

Level 3: new operators for repetition of sub-sequences

2.0 Metal Density

relation to ALF 2.0 9.2, 9.5

relation to ALF IEEE

History initial draft April 16 2001 by Wolfgang
reviewed and retained by Study Group April 16
o.k. as is by Tim Ehrler May 1

2.1 Motivation

Manufacturability in 130 nm technology and below requires so-called metal density rules. For a given routing layer, metal must cover a certain percentage of the total area within a lower and upper bound in order to ensure planarity. This percentage also depends on the total area under consideration, i.e., there are “local” and “global” metal density rules.

2.2 Proposal

Introduce new keyword DENSITY (or other word) for arithmetic model. Shall be non-negative number normalized between 0 and 1 (1 means 100%). Usable in context of LAYER (see ALF 2.0, chapter 9.5.1) with PURPOSE=routing (see ALF 2.0, chapter 9.5.2). Legal argument (i.e. HEADER) includes AREA, meaning the die area subjected to manufacturing of this layer.

Example:

```
LAYER metal1 {
   PURPOSE = routing;
   LIMIT {
      DENSITY {
         MIN {
            HEADER {
               AREA {
                  INTERPOLATION = floor;
                  TABLE { 0 100 1000 }
               }
            }
            TABLE { 0.2 0.3 0.4 }
         }
         MAX {
            HEADER {
               AREA {
                  INTERPOLATION = floor;
                  TABLE { 0 100 1000 }
               }
            }
            TABLE { 0.8 0.7 0.6 }
         }
      }
```

```
    }
}
```

Within an area of less than 100 units, the metal density must be between 20% and 80%. Within an area of 100 up to less than 1000 units, the metal density must be between 30% and 70%. Within an are of 1000 units or more, the metal density must be between 40% and 60%. The annotation INTERPOLATION=floor indicates that no interpolation is made for areas in-between, but the next lower value is used (see ALF 2.0, chapter 7.4.4).

3.0 Current

relation to ALF 2.0 8.1, 8.7, 8.15

relation to ALF IEEE

History initial draft April 162001 by Wolfgang
reviewed and retained by Study Group April 16
also reviewed by Tim Ehrler May 1
add text to clarify purpose by Wolfgang May 4
proposal reviewed May 8, added supplementary proposal

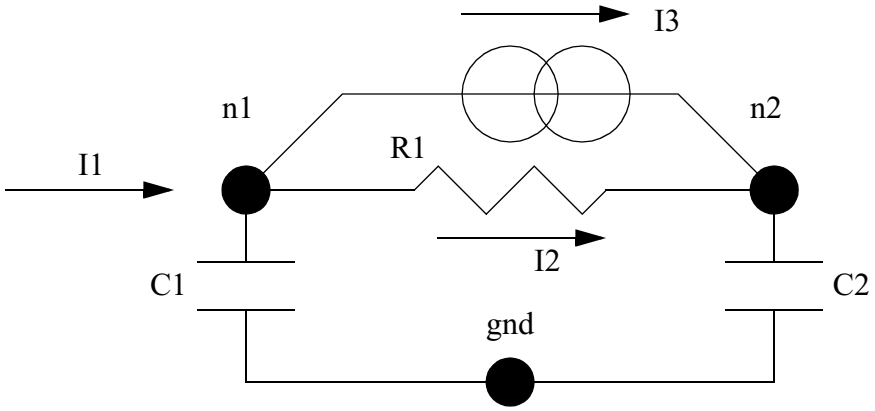
3.1 Motivation

CURRENT needs PIN annotation indicating the target point where the current is flowing into. Cannot define a branch of an electrical network where the current flows through.

Therefore there will be 3 types of CURRENT specification:

I1 = current into PIN from unspecified source (already supported in ALF 2.0)
I2 = current through a COMPONENT with two terminal nodes
I3 = current through an independent current source connected between two NODEs

see I1, I2, I3 in illustration



3.2 Proposal

In the context of WIRE, the following annotations for CURRENT shall be legal:

```
PIN = pin_identifier ;
```

Current flows from unknown source into the pin (already supported).

```
COMPONENT = component_identifier ;
```

Current flows through the component. The component must be a declared two-terminal electrical component in the context of the WIRE, i.e. a RESISTANCE, CAPACITANCE, VOLTAGE or INDUCTANCE (excluding mutual inductance, which has 4 terminals). The direction of the current flow is given by the order of node identifiers in the NODE annotation for that component (see ALF 2.0, chapter 8.15.3, 8.15.4).

```
NODE { 1st_node_identifier 2nd_node_identifier }
```

Current flows through a current source connected between the nodes. The direction of the current flow is given by the order of node identifiers in this NODE annotation.

Example:

```
WIRE interconnect_analysis_model_1 {
   CAPACITANCE C1 { NODE { n1 gnd } }
   CAPACITANCE C2 { NODE { n2 gnd } }
   RESISTANCE  R1 { NODE { n1 n2  } }
   CURRENT I1 { PIN = n1; }
   CURRENT I2 { COMPONENT = R1; }
   CURRENT I3 { NODE { n1 n2 } }
}
```

This example corresponds exactly to the illustration shown above.

3.3 Supplementary proposal

According to ALF 2.0, chapter 8.7.3, the sense of measurement for current associated with a pin shall be *into* the node. However, in some cases, the natural sense of measurement is *out of* the node. In order to allow explicit specification of the sense of measurement, the following feature is proposed:

FLOW annotation for current shall specify the sense of measurement of current. Default value shall be “in”, which is backward compatible with ALF 2.0.

```
FLOW = in | out;
```

For example, the following two statements are equivalent:

```
CURRENT I1 = 3.0 { PIN = n1; FLOW = in; }

CURRENT I1 = -3.0 { PIN = n1; FLOW = out; }
```

This is illustrated in the picture below.

4.0 Noise

relation to ALF 2.0 8.1, 8.14

relation to ALF IEEE

History initial draft April 16 2001 by Wolfgang
o.k by Tim Ehrler May 1
updated by Wolfgang May 4
reviewed and updated (see minutes) May 8

4.1 Motivation

NOISE_MARGIN defines a normalized voltage difference between nominal signal level and tolerated signal level. If violated, the correct signal level can not be determined. In order to check against noise margin, actual noise must be calculated. Currently VOLTAGE is used for noise calculations. However, since noise margin is normalized to signal voltage swing, it would be convenient, if the actual noise could also be represented in a normalized way. In CMOS, actual noise and noise margin tend to scale with supply voltage. A non-normalized model requires supply voltage as a parameter, if the supply voltage is subject to variation. A normalized model would to a 1st order degree approximate the voltage scaling effect already and therefore eliminate the supply voltage as a model parameter.

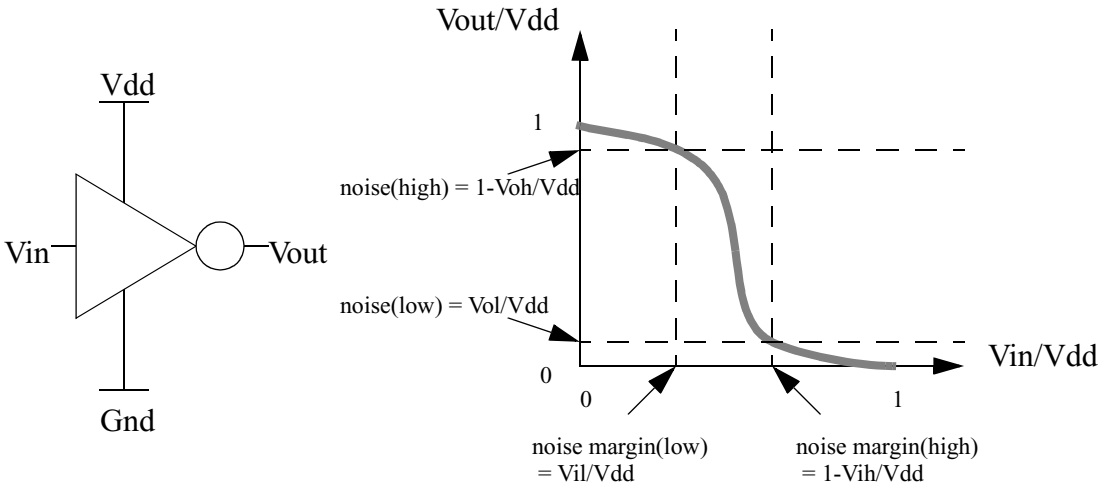
4.2 Proposal

Introduce new keyword NOISE, representing a normalized voltage difference between nominal signal level and actual signal level. Same measurement definition as for noise margin (see ALF 2.0, chapter 8.14). Noise margin is violated, if noise is larger than noise margin.

Context-specific meaning of NOISE

1. Context is output or bidirectional PIN

NOISE specifies maximum amount of noise at output pin, when any input pin is subjected to the amount of noise specified by NOISE_MARGIN. NOISE may have submodel HIGH and LOW. The relation between noise at output pin and noise margin at input pin is illustrated in the following picture.



Example:

```
PIN my_input_pin {
   DIRECTION = input;
   NOISE_MARGIN { HIGH = 0.3; LOW = 0.2; }
}
PIN my_output_pin {
   DIRECTION = output;
   NOISE { HIGH = 0.02; LOW = 0.01; }
}
```

2. Context is VECTOR with `vector_expression`

NOISE needs PIN annotation. NOISE specifies peak noise while pin is in “*” state. NOISE may only have submodel HIGH and LOW, if “?” state as opposed to “0” or “1” state is specified in vector_expression.

Example:

```
VECTOR ( 0* my_pin -> *0 my_pin) {
   NOISE = 0.2 { PIN = my_pin; }
}
```

3. Context is CELL, SUBLIBRARY, or LIBRARY

no PIN annotation. NOISE specifies maximum amount of noise at any output or bidirectional pin within scope, unless this specification is overwritten locally.

Example:

```
LIBRARY my_library {
   NOISE { HIGH = 0.02; LOW = 0.01; }
}
```

5.0 Non-scan cell

relation to ALF 2.0 6.2, 11.2

relation to ALF IEEE

History initial draft April 16 2001 by Wolfgang
o.k. by Tim Ehrler May 1

5.1 Motivation

Non-scan cell defines the mapping between the pins of a non-scan cell (left-hand side) and the pins of a scan cell (right-hand side). The scan cells has always certain pins which do not exist in the non-scan cell. In some cases, the non-scan cell might have certain pins which do not exist in the scan cell (In such a case, the scan replacement can only be done, if the pin in question was tied to an inactive level in the non-scan cell in the first place).

Currently, the non-scan cell statement supports definition of LHS or RHS constants which specify the logic level to which the non-corresponding pins should be tied to. However, this definition is redundant, because every relevant pin in a cell model must have annotations for SIGNALTYPE and POLARITY in order to be usable for DFT tools. These annotations specify already the logic level to which non-corresponding pins must be tied.

5.2 Proposal

Reduce syntax for pin_assignment (see ALF 2.0, chapter 11.2) to the following:

```
pin_assignment ::=
   pin_identifier [ index ] = pin_identifier [ index ] ;
|  pin_identifier [ index ] = logic_constant ;
```

Only “ `pin_identifier [ index ] = pin_identifier [ index ] ;` “ will actually be used for non-scan cell. Since POLARITY defines the active signal level, the pin should be tied to the opposite level. For pins without POLARITY, the level does not matter (e.g. scan input for scan flipflop in non-scan mode).

Example (taken from ALF 2.0, chapter 6.2):

```
CELL my_flipflop {
   PIN q   { DIRECTION=output; } // SIGNALTYPE defaults to “data”
   PIN d   { DIRECTION=input; } // SIGNALTYPE defaults to “data”
   PIN clk { DIRECTION=input; SIGNALTYPE=clock; POLARITY=rising_edge; }
   PIN clear { DIRECTION=input; SIGNALTYPE=clear; POLARITY=low; }
}
CELL my_scan_flipflop {
   PIN data_out { DIRECTION=output; } // SIGNALTYPE defaults to “data”
   PIN data_in  { DIRECTION=input; } // SIGNALTYPE defaults to “data”
   PIN scan_in  { DIRECTION=input; SIGNALTYPE=scan_data; }
   PIN scan_sel { DIRECTION=input; SIGNALTYPE=scan_control;
```

```
      POLARITY { SCAN=high; } } // scan mode when 1, non-scan mode when 0
   PIN clock {DIRECTION=input; SIGNALTYPE=clock; POLARITY=rising_edge;}
   NON_SCAN_CELL {
      my_flip_flop {
         clk = clock;
         d   = data_in;
         q   = data_out;
      }
   }
}
```

The scan replacement works only, if the **clear** pin of **my_flip_flop** is tied high (active level is low). Note: This is an exceptional case and only shown because it might happen eventually. Normally, the pins of the scan cell represent a superset of the pins of the non-scan cell.

In order to simulate the non-scan mode, when the non-scan cell is replaced by the scan cell, the **scan_sel** pin of **my_scan_flipflop** must be tied low (scan mode level is high). The **scan_in** pin can be tied to either high or low.

This example shows that the constant logic levels need not be defined in the non-scan cell statements, because they can be completely infered from the POLARITY statements. The POLARITY statements are mandatory for DFT tools anyway.

6.0 VIOLATION in context of LIMIT

relation to ALF 2.0 7.5, 7.6, 8.4

relation to ALF IEEE

History Proposal May 1 by Tim Ehrler
written in doc May 4 by Wolfgang
reviewed and updated (see minutes) May8

6.1 Motivation

Want to specify level of severity, if a LIMIT is violated. Target is appropriate error report from tool.

6.2 Proposal

The VIOLATION statement may appear within the context of an arithmetic model within LIMIT or an arithmetic submodel within LIMIT.

In this context, a MESSAGE_TYPE annotation or a MESSAGE annotation or both shall be legal within VIOLATION. A BEHAVIOR statement within VIOLATION shall only be legal if the LIMIT is within the context of a VECTOR. In the latter case, the `vector_expression` or `boolean_expression` which identifies the VECTOR shall define the triggering condition for the behavior described in the BEHAVIOR statement.

7.0 New value for MEASUREMENT annotation

relation to ALF 2.0 8.9.1

relation to ALF IEEE

History Proposal by Wolfgang, May 22

7.1 Motivation

Currently, measurements of analog quantities can be specified as “average”, “rms”, “peak”, “transient”, “static”. Another commonly used measurement is the average over absolute values, which cannot be specified.

7.2 Proposal

The MEASUREMENT annotation shall support the following values:

```
MEASUREMENT =
   transient
|  static
|  average
|  rms
|  peak
|  absolute_average[1]
```

The mathematical definition of **absolute_average** is the following:

$$\frac{\int_{(t=0)}^{(t=T)} |E(t)|\,dt}{T}$$

1. everything except **absolute_average** is already supported in ALF 2.0

8.0 MONITOR statement for VECTOR

relation to ALF 2.0 5.3.7, 5.4, 6.4.16

relation to ALF IEEE

History Proposal by Wolfgang, May 22

8.1 Motivation

Any `vector_expression` in the context of a VECTOR has an associated set of variables, which are monitored for the purpose of evaluating the `vector_expression`. The set of variables is given by the set of declared PINs, featuring a SCOPE annotation.

```
SCOPE = behavior | measure | both | none ; // see ALF 2.0, chapter 6.4.16
```

In the context of a VECTOR, all PINs with **`SCOPE = measure | both`** are monitored. Sometimes it would be practical to reduce the set of monitored pins within the scope of a particular vector. For example, in a multiport RAM, only the pins associated with a particular logical port should be monitored, if the `vector_expression` describes a transaction involving only this port. Currently, this can only be achieved by applying the "?*" operator to all unmonitored pins. Therefore the `vector_expression` can become quite lengthy for complex cells.

8.2 Proposal

A VECTOR identified by a `vector_expression` may have the following MONITOR annotation:

```
monitor_multivalue_annotation :==
   MONITOR { pin_identifiers }
```

The set of *`pin_identifiers`* shall be a subset of pins with **`SCOPE = measure | both`** .

If the MONITOR annotation is present, all pins appearing within this annotation shall be monitored. Any pin appearing in the `vector_expression` must also appear in the MONITOR annotation. However, all pins appearing in the MONITOR annotation need not appear in the `vector_expression`.

If the MONITOR annotation is not present, all pins with **`SCOPE = measure | both`** shall be monitored (backward compatible with ALF 2.0).

Example:

```
CELL my_4_bit_register_file {
   PIN clk { DIRECTION=input; }
   PIN [4:1] din  { DIRECTION=input; }
   PIN [4:1] dout { DIRECTION=output; }
   VECTOR ( 01 clk -> ?! dout[1] ) {
      MONITOR { din[1] dout[1] clk } // put in delay, power etc.
   }
   VECTOR ( 01 clk -> ?! dout[2] ) {
      MONITOR { din[2] dout[2] clk } // put in delay, power etc.
   }
   VECTOR ( 01 clk -> ?! dout[3] ) {
      MONITOR { din[3] dout[3] clk } // put in delay, power etc.
   }
   VECTOR ( 01 clk -> ?! dout[4] ) {
      MONITOR { din[4] dout[4] clk } // put in delay, power etc.
   }
}
```

9.0 Make grammar more compact by removing redundancies

relation to ALF 2.0 3.2, 11.x

relation to ALF IEEE

History Proposal by Wolfgang, May 22

9.1 Motivation

Simplify the grammr by getting rid of redundant definitions. Definitions which are used in a particular context should be presented in that context. This will also simplify to introduce grammar “sniplets” in the semantic sections, where they are needed.

9.2 Proposal

chapter 11.2

Get rid of chapter 11.2 and introduce the pertinent statements locally, where they are needed.

`unnamed_assignment_base`
remove

`unnamed_assignment`
rename to `single_value_annotation`, move to 11.7

`named_assignment_base`
remove

`named_assignment`
remove

```
single_value_assignment ::=
   identifier = value ;
```

`multi_value_assignment`
rename to `multi_value_annotation`, move to 11.7

`assignment`
remove

`pin_assignment`
modify according to chapter 5 of this doc., move to 11.7

`arithmetic_assignment`
move to 11.7

chapter 11.3

split into 3 separate chapters:

- Boolean expressions and operators
  put `boolean_expression`
- Arithmetic expressions and operators
  put `arithmetic_expression`
- Vector expressions and operators
  put everything else

chapter 11.4

Get rid of chapter 11.4 and introduce the pertinent statements locally, where they are needed.

`cell_instantiation`
remove

`unnamed_cell_instantiation`
only used for NON_SCAN_CELL, move to 11.9

`named_cell_instantiation`
only used for STRUCTURE, move to 11.17

`pin_instantiation`
only used for PIN, move to 11.11

Error to be corrected:
incorrect use of `pin_instantiation` in chapter 6.3, should be `pin_assignments`

Error to be correted:
`pin_instantiation` is not mentioned as `pin_item` in chapter 11.11

`primitive_instantiation`
only used for FUNCTION, move to 11.17

`template_instantiation`
move to 11.7

`dynamic_instantiation_item`
move to 11.7

`via_instantiation`
move to 11.23

chapter 11.5

move to “lexical rules” section (chapter 10)

chapter 11.6

Get rid of chapter 11.6, associate operators with the corresponding expressions.

- Boolean expressions and operators
  put all operators with prefix `boolean_`
- Arithmetic expressions and operators
  put all operators with prefix `arithmetic_`
- Vector expressions and operators
  put all operators with prefix `vector_`

move `sequential_if`, `sequential_else_if` to 11.17.

chapter 11.7

rename `logic_assignment` (see 11.17) into `boolean_assignment` and move into 11.7. Move `vector_assignment` into 11.7.

rewrite grammar involving `all_purpose_item`, `annotation`, `annotation_container`, the other items remain unchanged.

```
annotation ::=
   one_level_annotation
|  two_level_annotation
|  multi_level_annotation

one_level_annotation ::=
   single_value_annotation
|  multi_value_annotation

one_level_annotations ::=
   one_level_annotation { one_level_annotation }

two_level_annotation ::=
   one_level_annotation
|  identifier [ = value ] { one_level_annotations }

two_level_annotations ::=
   two_level_annotation { two_level_annotation }

multi_level_annotation ::=
   one_level_annotation
|  identifier [ = value ] { multi_level_annotations }

multi_level_annotations ::=
   multi_level_annotation { multi_level_annotation }

annotation_container ::=
   identifier { one_level_annotations }
```

Since all_purpose_item allows generic_object and generic_object includes keyword_declaration statement, consequently all *syntax_item*_identifiers that can be used by keyword_declaration (see chapter 3.2.9) must be covered by all_purpose_item.

```
all_purpose_item ::=
   generic_object
|  template_instantiation
|  annotation
|  arithmetic_model
|  arithmetic_model_container
|  boolean_assignment
|  vector_assignment
```

Error to be correted:
boolean_assignment is not mentioned as *syntax_item*_identifier in chapter 3.2.9.

Note: arithmetic_submodel is also a *syntax_item*_identifier, but it is not included in all_purpose_item, because arithmetic_submodel is always in the context of arithmetic_model.

10.0 Rewrite grammar for more specific syntax and less semantic restriction

relation to ALF 2.0 3.2, 11.x

relation to ALF IEEE

History Proposal by Wolfgang, May 22

10.1 Motivation

Certain syntax definitons af ALF are written in a very generic way. As a consequence, a lot of semantic restrictions apply. The idea is to rewrite the grammar so that the syntax section becomes more specific and as a consequence the semantic sections become less “heavy”. However, the changes to the existing grammar should be limited to modifications which specifically serve that purpose rather than re-writing the whole grammar from scratch. Also, eventual redundancy in the grammar can be eliminated.

10.2 Proposal

Use `all_purpose_item` only for statements with custom keywords, introduced by `keyword_declaration` statements and put the statements using standard keywords explicitely in the grammar.

Chapter 11.9

```
cell_item ::=
   all_purpose_item
|  CELLTYPE_single_value_annotation
|  SWAP_CLASS_one_level_annotation
|  RESTRICT_CLASS_one_level_annotation
|  SCANTYPE_single_value_annotation
|  SCAN_USAGE_single_value_annotation
|  BUFFERTYPE_single_value_annotation
|  DRIVERTYPE_single_value_annotation
|  PARALLEL_DRIVE_single_value_annotation
|  pin
|  pin_group
|  primitive
|  function
|  non_scan_cell
|  test
|  vector
|  wire
|  blockage
|  artwork
|  connectivity
```

Chapter 11.10

```
library_item ::=
   all_purpose_item
```

Chapter 11.11

```
pin_item ::=
   all_purpose_item
|  range
|  VIEW_single_value_annotation
|  PINTYPE_single_value_annotation
|  DIRECTION_single_value_annotation
|  SIGNALTYPE_single_value_annotation
|  ACTION_single_value_annotation
|  POLARITY_two_level_annotation
|  DATATYPE_single_value_annotation
|  INITIAL_VALUE_single_value_annotation
|  SCAN_POSITION_single_value_annotation
|  STUCK_single_value_annotation
|  SUPPLYTYPE_single_value_annotation
|  SIGNAL_CLASS_one_level_annotation
|  SUPPLY_CLASS_one_level_annotation
|  cell_pin_reference_two_level_annotation
|  DRIVETYPE_single_value_annotation
|  SCOPE_single_value_annotation
|  PULL_single_value_annotation
|  port
|  connectivity
|  pin_instantiation // this one is missing in chapter 11.11
```

Chapter 11.14

```
vector_item ::=
   all_purpose_item
|  PURPOSE_one_level_annotation
|  OPERATION_single_value_annotation
|  LABEL_single_value_annotation
|  EXISTENCE_CLASS_one_level_annotation
|  EXISTENCE_CONDITION_boolean_assignment
|  CHARACTERIZATION_CLASS_one_level_annotation
|  CHARACTERIZATION_CONDITION_boolean_assignment
|  CHARACTERIZATION_VECTOR_vector_assignment
|  MONITOR_one_level_annotation // proposed in this doc chapter 8
|  illegal_statement
```

Chapter 11.15

```
wire_item ::=
   all_purpose_item
|  SELECT_CLASS_one_level_annotation
|  node

node ::=
   NODE name_identifier { node_items }
```

```
node_items ::=
   node_item { node_item }

node_item ::=
   all_purpose_item
|  NODETYPE_single_value_annotation
|  NODE_CLASS_one_level_annotation
```

Chaper 11.16

```
arithmetic_models ::=
   arithmetic_model { arithmetic_model }

arithmetic_model ::=
   partial_arithmetic_model
|  full_arithmetic_model
```

Partial arithmetic model contains only definitions, no data. Can appear outside the semantically valid context of the model, as long as a semantically valid context exists within scope. (Example: semantically valid context of arithmetic model X is VECTOR, VECTOR exists within scope of LIBRARY, therefore partial arithmetic model X is legal within LIBRARY.) Definitions inside partial arithmetic model without *name*_identifier are inherited by each arithmetic model with *arithmetic_model*_identifier within scope. (Note: up to 2 levels of submodel are supported)

```
partial_arithmetic_model ::=
   arithmetic_model_identifier [ name_identifier ] {
      { all_purpose_item }
      { arithmetic_model_qualifier }
      { partial_arithmetic_submodel }
   }

partial_arithmetic_submodel ::=
   arithmetic_submodel_identifier [ name_identifier ] {
      { all_purpose_item }
      { partial_arithmetic_leaf_submodel }
   }

partial_arithmetic_leaf_submodel ::=
   arithmetic_submodel_identifier [ name_identifier ] {
      { all_purpose_item }
   }
```

Full arithmetic model contains both definitions and data. Can only appear in the semantically valid context of the model. Enables evaluation of arithmetic model in design context (e.g. delay calculation, power calculation). A trivial arithmetic model contains directly the evaluation value. A non-trivial arithmetic model requires calculation of the value, based on evaluation conditions. (Note: up to 2 levels of submodel are supported)

```
full_arithmetic_model ::=
   trivial_arithmetic_model
|  non_trivial_arithmetic_model
```

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

non_trivial_arithmetic_model ::=
   arithmetic_model_identifier [ name_identifier ] {
      { all_purpose_item }
      { arithmetic_model_qualifier }
      arithmetic_model_body
      { arithmetic_model_datarange }
   }
|  arithmetic_model_identifier [ name_identifier ] {
      { all_purpose_item }
      [ violation ]
      { arithmetic_model_qualifier }
      full_arithmetic_submodels
   }

full_arithmetic_submodels ::=
   full_arithmetic_submodel { full_arithmetic_submodel }

full_arithmetic_submodel ::=
   full_arithmetic_leaf_submodel
|  arithmetic_submodel_identifier [ name_identifier ] {
      { all_purpose_item }
      full_arithmetic_leaf_submodels
   }

full_arithmetic_leaf_submodels ::=
   full_arithmetic_leaf_submodel { full_arithmetic_leaf_submodel }

full_arithmetic_leaf_submodel ::=
   trivial_arithmetic_leaf_submodel
|  non_trivial_arithmetic_leaf_submodel

trivial_arithmetic_leaf_submodel ::=
   arithmetic_submodel_identifier [ name_identifier ] = value ;
|  arithmetic_submodel_identifier [ name_identifier ] = value {
      { all_purpose_item }
   }

non_trivial_arithmetic_leaf_submodel ::=
   arithmetic_submodel_identifier [ name_identifier ] {
      { all_purpose_item }
      arithmetic_model_body
      { arithmetic_model_datarange }
   }
```

Auxiliary definitions for arithmetic model. Semantic restrictions apply. (Note: the new grammar allows non-ambiguous distinction between usage of MIN/TYP/MAX/ DEFAULT either as `arithmetic_leaf_submodel` or as `single_value_annotation`.)

```
arithmetic_model_qualifier ::=
   general_arithmetic_model_qualifier
|  connected_arithmetic_model_qualifier
|  analog_arithmetic_model_qualifier
|  timing_arithmetic_model_qualifier
|  layout_arithmetic_model_qualifier

general_arithmetic_model_qualifier ::=
   UNIT_single_value_annotation
|  CALCULATION_single_value_annotation
|  INTERPOLATION_single_value_annotation

connected_arithmetic_model_qualifier ::=
   PIN_one_level_annotation
|  NODE_one_level_annotation

analog_arithmetic_model_qualifier ::=
   analog_MEASUREMENT_single_value_annotation
|  COMPONENT_single_value_annotation
|  TIME_arithmetic_model
|  FREQUENCY_arithmetic_model

timing_arithmetic_model_qualifier ::=
   EDGE_NUMBER_single_value_annotation
|  violation
|  from
|  to

layout_arithmetic_model_qualifier ::=
   distance_MEASUREMENT_single_value_annotation
|  BETWEEN_multi_value_annotation
|  REFERENCE_single_value_annotation
|  ANTENNA_one_level_annotation
|  PATTERN_single_value_annotation
|  VIA_single_value_annotation

arithmetic_model_datarange ::=
   MIN_single_value_annotation
|  TYP_single_value_annotation
|  MAX_single_value_annotation
|  DEFAULT_single_value_annotation

arithmetic_model_body ::=
   [ header ] table [ equation ]
|  [ header ] equation [ table ]

equation ::=
   EQUATION { arithmetic_expression }

from ::=
   FROM {
      [ PIN_single_value_annotation ]
      [ EDGE_NUMBER_single_value_annotation ]
      [ THRESHOLD_arithmetic_model ]
   }
```

```
to ::=
   TO {
      [ PIN_single_value_annotation ]
      [ EDGE_NUMBER_single_value_annotation ]
      [ THRESHOLD_arithmetic_model ]
   }
```

Auxiliary definitions for arithmetic model, also applicable elsewhere (separate chapter?).

VIOLATION is also applicable for ILLEGAL

```
violation ::=
   VIOLATION {
      [ MESSAGE_TYPE_single_value_annotation ]
      [ MESSAGE_single_value_annotation ]
      [ behavior ]
   }
```

TABLE and HEADER are also applicable for CONNECTIVITY.

```
table ::=
   TABLE { values }

header ::=
   HEADER { arithmetic_model_identifiers }
   HEADER { header_arithmetic_models }
```

Arithmetic model in context of HEADER (Note: no submodels allowed).

```
header_arithmetic_models ::=
   header_arithmetic_model { header_arithmetic_model }

header_arithmetic_model ::=
   arithmetic_model_identifier [ name_identifier ] {
      { all_purpose_item }
      { arithmetic_model_qualifier }
      { arithmetic_model_body }
      { arithmetic_model_datarange }
   }
```

Container of arithmetic model (Note: LIMIT is special).

```
arithmetic_model_containers ::=
   arithmetic_model_container { arithmetic_model_container }

arithmetic_model_container ::=
   limit
|  arithmetic_model_container_identifier { arithmetic_models }

limit ::=
   LIMIT { limit_arithmetic_models }
```

Arithmetic model in context of LIMIT (Note: must contain leaf submodels MIN and/or MAX).

```
limit_arithmetic_models ::=
   limit_arithmetic_model { limit_arithmetic_model }

limit_arithmetic_model ::=
   arithmetic_model_identifier [ name_identifier ] {
      { all_purpose_item }
      [ violation ]
      { arithmetic_model_qualifier }
      limit_arithmetic_submodels
   }

limit_arithmetic_submodels ::=
   limit_arithmetic_submodel { limit_arithmetic_submodel }

limit_arithmetic_submodel ::=
   limit_leaf_arithmetic_submodel
|  arithmetic_submodel_identifier [ name_identifier ]  {
      { all_purpose_item }
      [ violation ]
      limit_arithmetic_leaf_submodels
   }

limit_arithmetic_leaf_submodels ::=
   limit_arithmetic_leaf_submodel { limit_arithmetic_leaf_submodel }

limit_arithmetic_leaf_submodel ::=
   min_or_max = number ;
|  min_or_max {
      { all_purpose_item }
      [ violation ]
      [ arithmetic_model_body ]
   }

min_or_max ::=
   MIN
|  MAX
```

11.0 Item

relation to ALF 2.0 3.2.9, 11.x

relation to ALF IEEE reference to ALF IEEE chapter

History date of initial draft, date of revisions

11.1 Motivation

The idea is to define pertinent features of ALF using the ALF language itself. Such a definition could be used as a standard “header” file for ALF. Eventually, certain extensions of the language could then be defined by changing the header file instead of changing the language. This can be used for pure documentation purpose as well as for development of self-adapting ALF parsers.

11.2 Proposal

Use the KEYWORD statement to define standard arithmetic models.

Use the `definition_for_arithmetic_model` construct to define legal statements in the context of arithmetic models.

Use the CLASS statement for shared definitions.

Example (just to show the idea):

```
KEYWORD PROCESS = arithmetic_model ;
KEYWORD SLEWRATE = arithmetic_model ;
KEYWORD CURRENT = arithmetic_model ;

PROCESS {
   TABLE { nom spsn spwn wpsn wpwn }
}
CLASS all_models {
   KEYWORD UNIT = single_value_annotation ;
}
CLASS timing_models {
   CLASS { all_models }
   UNIT = 1e-9 ;
   KEYWORD RISE = arithmetic_model ;
   KEYWORD FALL = arithmetic_model ;
}
CLASS analog_models {
   CLASS { all_models }
   KEYWORD MEASUREMENT = single_value_annotation ;
}
SLEWRATE {
   CLASS { timing_models }
}
```

```
CURRENT {
   CLASS { analog_models }
   UNIT = 1e-3 ;
}
```

It may be worthwhile to explore how far we can get in describing ALF features in this language.