

IBIS Macro Language Reference Manual

I/O Buffer Information Specification Macro Language (IBIS-ML)
Language Reference Manual Version 05.

The IBIS Macro Language is a standard language suitable for modeling the analog characteristics of a digital integrated circuit. The IBIS Macro Language is intended for use in conjunction with the ANSI/EIA-656A IBIS standard.

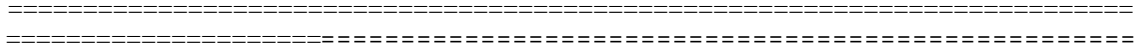
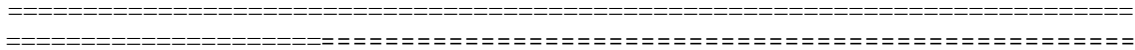


TABLE OF CONTENTS



G E N E R A L O V E R V I E W..... 6

S T A T E M E N T O F I N T E N T..... 7

 1.1. Background 7

 1.2. Relationship between IBIS and IBIS-ML..... 8

 1.3. Commitment to Backwards Compatibility 8

S Y N T A X R U L E S A N D G U I D E L I N E S 9

 1.1. Typographical Conventions 9

 1.2. Definition of Terms..... 9

 1.3. Syntax Rules 9

 1.4. Reserved Words 10

G E N E R A L O V E R V I E W 12

 4.1. Introduction..... 12

 4.2. Creating new Model_Types Using the [Define] Keyword 12

 4.3. An IBIS-ML Example 13

 4.4. IBIS-ML Library Files 15

B A S I C C I R C U I T E L E M E N T S 17

 Basic Circuit Elements..... 17

 Instantiating Circuit Elements..... 18

 Basic Element Descriptions 19

 The tline Element 23

 Complex Elements 25

 Assigning Values To An Element 26

 Subcircuits..... 28

**E X P R E S S I O N S , F U N C T I O N S , T A B L E S , A N D M A T R I C
E S..... 30**

 5.1. Symbolic Values 30

 5.2. Expressions 30

 5.3. Tables..... 31

 5.4. Matrices..... 31

 5.5. Functions 31

T E S T A N D C O N T R O L C O N S T R U C T S..... 32

 6.1. The “assert” Statement..... 33

- 6.2. The "define" Statement 34
- 6.3. The "foreach" Statement 35
- 6.4. The "if" Statement..... 37
- 6.5. The "inherit" Statement..... 39
- 6.6. The "node" Statement 40
- 6.7. The "select" Control Statement 41

- D A T A T Y P E S..... 45**
 - 7.1. Strings 45
 - 7.2. Scalars 46
 - 7.3. Tables..... 47
 - 7.4. Multi-dimensional tables 49
 - 7.5. Scopes 51
 - 7.6. Foreach arrays 53

- F U T U R E E X T E N S I O N S..... 55**

- temporary use only..... 56**

=====

REVISION HISTORY

=====

10/22/01 – Stephen Peters

- Rev 0.5
- Updated/expanded information on element (section 5). Added details on trigger, submodel
- Updated and expanded element and t-line descriptions
- Cleaned up section titles, however, messed up auto-numbering
- Updated example in section 4 (selection of power supply rails per AI's 3.2 examples)
- Added ! (bang) to list of reserved words

10/17/01 – Lynne Green

- Rev 0.4
- Split section 1 into 2 sections, based on Stephen's draft, updated section numbering to match
- Removed stand-alone [Define] Section (will be covered in main IBIS spec)
- Cleanup of syntax based on committee meeting of Oct 11.
- Modified Section 3, added Stephen's stuff, removed reference to all pre-defined constants.
- Need to fix all examples, if we agree on "typographical conventions". Courier for examples also recommended, along with italics.

10/9/01 – Lynne Green

- Rev 0.3c
- Added format for tlines.
- Cleanup of Sections 2, 4-5.2, based on committee meeting of Oct 4.
- Put Section 3: The Define Keyword, back in.
- Turned autonumbering on for the sections. Regenerated TOC.

09/7/01 – Lynne Green

- Rev 0.3b
- Removed references to IBIS-X specifically, replaced with reference to IBIS or EIA/IBIS specification.
- Complete rewrite of Section 2, based on Stephen's draft & committee comments.
- Rewrite of section 3.1.
- Rewrite of section 3.2 (first model example).
- Cleanup up expression and table formatting. Fixed case for Time, Temperature, Frequency.
- Still need format for tlines, consistent use of Courier for ML statements.

08/31/01 – Lynne Green

- Rev 0.3a
- Merged Sections 1 and 2.
- Removing things not specific to the macro language. (Moved this stuff to new Section for now.)

08/10/01 – Lynne Green

- Rev 0.2c
- Updated primitives list and formats, based on meeting of 7/31/01
- Created text in Section 6.

- Section 3: Started a list of reserved words.
- Section 9: Started a list of "Future Extensions".
- Examples and explanatory text still need to be checked for consistent Case use.

06/19/01 -- Stephen Peters

- Rev 0.2b
- Add some basic formatting, sections 7 and 8 work
- Start on section 5, still very much work in progress

11/15/2000 -- Stephen Peters

- Initial Revision 0.1

Section 1

GENERAL OVERVIEW

This section gives a general overview of the rest of the document.

Section 2 contains general information about the IBIS Macro Language (IBIS-ML) and its relationship to IBIS. Section 3 presents the general syntax rules and guidelines for creating an IBIS-ML file.

The formal specification of IBIS-ML follows in sections 4 through 8. Section 4 presents a general overview of the language while section 5 describes the circuit elements and nodal netlist syntax. Section 6 concerns itself with expressions. IBIS-ML control keywords and circuit building control constructs are detailed in section 7. Finally, section 8 discusses data types and the rules for creating and using the data templates in the IBIS models.

Section 9 describes items that were discussed, and that may be addressed in future versions. These items are not part of the 1.0 IBIS Macro Language Version 1.0 specification.

Section 2

STATEMENT OF INTENT

This document is the language reference manual (LRM) for the IBIS Macro Language (IBIS-ML).

The purpose of the IBIS macro language is to give model providers the ability to specify how an EDA tool (specifically, a circuit or transmission line simulator) is to use the data contained in an IBIS file. In other words, the macro language allows a model provider to create a behavioral level description of a digital I/O buffer's operation. An IBIS-ML description, together with the IBIS data template, is used to electronically transport I/O buffer model data between semiconductor vendors, simulation vendors, and end customers.

1.1. Background

Current CAE tools developed specifically for signal integrity simulation and analysis apply a set of well defined and understood data to standard simulation algorithms. These algorithms, in turn, are based on a standard I/O buffer model topology. In other words, it is assumed that the I/O buffers used for digital logic can be abstracted to a set of controlled voltage and current sources connected in a more-or-less fixed manner. Given this fixed model, the user needed only to supply specific data for each I/O buffer. This data consists primarily of the DC I/V (output voltage vs. current) characteristics, plus V/T (output voltage vs. time) waveforms that describe an outputs transition from one state to another. In addition, other "data book" type information is included so as to allow for system level timing analysis and error checking.

The original IBIS specification was the first industry standard method for electronically transporting I/O buffer modeling data. However, while the original IBIS specification has obtained wide industry acceptance, it has become increasingly clear that the original paradigm of data applied to a *fixed* model is no longer able to fully meet the needs of the industry. This is due to a number of issues, some of which are:

- Section 1 The increasing complexity of I/O buffers
- Section 2 The advent of new signaling technologies
- Section 3 Simulation methodologies that require describing the input-to-output relationships of a digital receiver, and finally
- Section 4 The need to accurately model the effects of on-die power distribution, return paths, and pin to pin coupling on an IO pin's behavior.

Therefore, the extended I/O Buffer Information Specification (IBIS) is proposed, along with the IBIS-ML macro language.

1.2. Relationship between IBIS and IBIS-ML

One of the fundamental concepts behind IBIS is the separation of data and algorithm. I/O buffer modeling data is contained in an IBIS file proper, while (as described above) the algorithm(s) that operated on the data were encoded in an EDA tool and could not be changed by the user. Now, with the definition of IBIS-ML, this separation of data and algorithm becomes explicit. IBIS data files contain model data – simulation data as well as ‘data book’ specifications – while the corresponding IBIS-ML file defines the behavior and usage of models (and other objects) that are referenced in the IBIS file. Note that both types of files are still considered ‘IBIS’ files and both use the ‘.ibs’ extension.

1.3. Commitment to Backwards Compatibility

 Section 3

 SYNTAX RULES AND GUIDELINES

This section contains general syntax rules and guidelines for IBIS-ML files as well as definitions of common terms and a list of reserved words. The document's typographical conventions are also explained.

1.1. Typographical Conventions

Examples and code fragments used in this document will follow these typographical conventions:

Verbatim text	appears in lowercase and must be entered exactly as shown
<i>italics</i>	indicates a placeholder. Substitute a literal
<item>	angle brackets indicate that the enclosed literal or word is optional
<item item>	single vertical bar indicates a choice between the listed items

1.2. Definition of Terms

symbolic constant	A symbolic constant is a placeholder in an IBIS-ML expression that, when evaluated, is replaced by the actual value of a [Model] sub-parameter, table, or keyword of the same name.
circuit element	One of the primitive elements (resistor, capacitor, etc.) used to build more complicated circuits.
element	short for circuit element.
[Model]	refers specifically to the IBIS notion of a model, as defined by the IBIS keyword, its syntax and usage.
simulation object	A [Model] and its associated [Define Model], representing a circuit topology (such as a buffer topology) and the information describing the circuit elements (that is, their values).

1.3. Syntax Rules

1) Unless overridden by a rule stated below, IBIS-ML shall follow the general syntax rules and guidelines as stated in the latest ANSI/EIA-656 (IBIS) specification.

Simulator state variables: Time, Temperature, Frequency
Netlist Operators: OPEN, SHORT
Circuit Parameters: Z0, TD, LEN, R, L, G, C, V, I, RlcgFile

Note: Because these words are reserved, they are case-insensitive. The capitalization shown is the preferred convention.

4.3. An IBIS-ML Example

Following is an example showing how a new `model_type` called 'simple_receiver' might be constructed. The purpose of this example is to introduce the reader to the syntax and structure of the language *as a whole* so that the reader may gain an overall understanding and 'feel' for the language before diving into the details presented in the later sections.

Let's assume that `simple_receiver` has conventional power and ground clamps plus an optional pullup resistor to the power rail. The model creator decides that the circuit (behavioral description) will include the clamps, the pullup resistor, and the input pad capacitance. In addition, to support various types of circuit and transmission simulators, the model creator allows the tool to choose between using a fixed power rail voltage supplied by a keyword or using a power rail voltage supplied via an external port. The corresponding `.ibs` data file must then contain I/V curves for the clamps, the value of the pullup resistor, the value of the pad capacitance, and a keyword specifying the models operating voltage range.

The receiver function described above can be expressed using the IBIS-ML as follows:

```
[Define Model] simple_receiver (vcc, gnd, pad)
|
|
| circuit description
capacitor io_cap (pad gnd) C=C_comp
resistor term_res (pad vcc) R=(R_pullup || OPEN)
iconrolled pclamp (pad vcc)(pad vcc) I=[POWER Clamp] (v(pad,vcc))
iconrolled gclamp (pad gnd)(pad gnd) I=[GND Clamp] (v(pad,gnd))
|
| optionally connects the vcc port to the voltage specified
| by the [Voltage Range] keyword
|
if (!vcc)
  vsource power_supply (vcc gnd) V=[Voltage Range]
endif
|
[End Define Model] simple_receiver
```

For reference, the corresponding IBIS [Model] keyword for `simple_receiver` is as follows:

```
[Model] xyz
model_type simple_receiver
|
|
C_comp      3pF  2pF  4pF
R_pullup    50   47   55
|
|variable          typ  min  max
[Voltage Range]  3.3v 3.0v 3.6v
|
[Power Clamp]
| volts  I(typ)  I(min)  I(max)
-3.3v   -100ma  -80ma   -120ma
-2.5v           <etc>
|
[Gnd Clamp]
| volts  I(typ)  I(min)  I(max)
```

```
-3.3v    100ma    80ma    120ma
-2.5v          <etc>
```

The [Model] keyword contains a set of data specific to instance ‘xyz’ of model_type simple buffer.

The IBIS-ML description of the behavior of simple_receiver begins with the line

```
[Define Model] simple_receiver (vcc, gnd, pad)
```

The [Define Model] keyword informs the EDA tool that a new simulation object (in this case an IBIS [Model] with the model_type of ‘simple receiver’) is being defined. This model includes an optional port list of (vcc, gnd, pad). Note that a model_type must be defined before it can be used. In other words, the [Define] keyword that defines a model_type must appear in the IBIS file before the [Model] keyword that uses that model_type. All IBIS-ML descriptions begin with a [Define] keyword.

Note: For a complete description of the [Define] keyword syntax and usage, as well as a definition of object classes and types, refer to the IBIS specification.

In general, I/O buffer models such as simple_receiver are built from basic circuit elements surrounded by test-and-control construct. Each circuit element has an element name (capacitor), an instance name (io_cap), nodes (pad gnd) and a value (C_comp). Circuit element values can be symbolic constants (C_comp), fixed values (3.3v), or tables ([GND Clamp]). Test-and-control constructs test for the existence or value of keywords or sub-parameters and then alter the circuit description based on the results of the test.

We are now ready to look in detail at each IBIS-ML statement in the [Define Model] section. The actual circuit description of simple_buffer begins with the line.

```
capacitor io_cap (pad gnd) C=C_comp
```

This line instructs the EDA tool to include a capacitor with the name ‘io_cap’ between the nodes ‘pad’ and ‘gnd’ of the [Model] simple_receiver. The capacitor value is assigned (C=) the value of the symbolic constant ‘C_comp’. In this example, the sub-parameter ‘C_comp’ under the [Model] keyword supplies the value for the symbolic constant ‘C_comp’ in the [Define Model] section. In IBIS-ML descriptions, symbolic constants are placeholders for values (strings, numbers, arrays of values, etc.), where the specific values are given in the data section.

```
[Model] xyz
model_type simple_input
.
.
C_comp 3pF 2pF 4pF
.
.
```

The next line of IBIS-ML defines a resistor named term_res, between the nodes pad and vcc.

```
resistor term_res (pad vcc) R=(R_pullup || OPEN)
```

The resistor value is assigned (R=) the value of the symbolic constant 'R_pullup', as specified by R= (R_pullup || OPEN). This is read, "R equals the value of R_pullup or, if the R_pullup sub-parameter is not present in the [Model], then R equals OPEN", where "OPEN" is an open circuit. This construct allows a user to specify an optional parameter.

The next line defines a controlled current source named pclamp. This controlled source represents the receiver's power clamp diode.

```
icontrolled pclamp (pad vcc)(pad vcc) I=[POWER Clamp] (v(pad,pwr))
```

The controlled source is connected between the nodes pad and vcc, and uses the voltages at the nodes in the second set of parentheses ("pad" and "vcc") as the controlling nodes. The term I= indicates that the source produces a current. The table name is [Power Clamp]. The table name matches the table name under the [Model] keyword, including the brackets. The table name is followed by the table input lookup values, enclosed in parentheses.

The next line describes a second controlled current source. This is the other clamp, and it uses a different control node pair and a different table from the first clamp.

```
icontrolled gclamp (pad gnd)(pad gnd) I=[GND Clamp] (v(pad,gnd))
```

The next three lines of the IBIS-ML description of simple_receiver implement the optional power supply connections.

```
if (!vcc)
  node vcc
  vsource power_supply (vcc gnd) V=[Voltage Range]
endif
```

The first line is an 'if' statement that read "if the node vcc is not connected externally, then do the following". Note that the 'bang' operator (!) is not just a simple existence test for the node vcc, it is testing to see whether or not the node is connected to an external circuit. If the test is true (i.e. vcc is not connected externally) then the node vcc is created and an independent voltage source named 'power_supply' connected between the nodes 'vcc' and 'gnd'. The independent source is assigned the value of the [Voltage Range] keyword (V=[Voltage Range]. If vcc were connected externally, then the conditional statement would fail and model power would be supplied by an external network thru the vcc port.

The [End Define] keyword terminates the description of simple_receiver:

```
[End Define Model] simple_receiver
```

4.4. IBIS-ML Library Files

The above model description may be included in a complete IBIS file describing a component and package, or a user may collect one or more IBIS-ML descriptions and place them in a stand alone text file with an .ibs extension. A file of this type is referred to as an IBIS-ML *library file*. A library file can be included in another IBIS file by use of the [Include Library] keyword. As with any IBIS compatible file an IBIS-ML library file must begin with a legal header section (as

defined by the IBIS specification) and is terminated with the [End] keyword. Between the end of the header section and the [End] keyword are the [Define]/[End Define] keywords for each object defined as well as the IBIS-ML statements themselves. With the exception of the keyword [Comment Char], a library file may not contain any keywords associated with a component or package section.

For reference, following is a tree diagram outlining the contents of a IBIS-ML library file. Both the required and optional keywords of the header sections are shown. The [Define "class name"/[End "class name"] keyword pair is repeated for each new model defined.

```
/-- Start of File
|-- optional comment text
|-- [IBIS Ver]
|-- [File Name]
|-- [File Revision]
|-- [Date]
|-- [Source]
|-- [Notes]
|-- [Disclaimer]
|-- [Copyright]
|
|  /--[Define "class name"]
|  |-- IBIS-ML statements
|  \--[End Define "class name"]
|
\--[End]
```

Section 5

BASIC CIRCUIT ELEMENTS

This section of the LRM describes the circuit elements that can be used in a [Define Model] template to create a model topology (behavioral circuit description). This section also describes the nodal syntax used for interconnecting circuit components into behavioral models

The primitive components include the traditional resistance, inductance, capacitance, independent and controlled source components, and transmission lines. By allowing these components to be described using tables and equations as well as traditional values, this set of components can support all IBIS constructs.

Basic Circuit Elements

Table 1 below lists the basic circuit elements, along with a brief description of each. For each element, the value may be an expression or table involving constants, symbolic constants, node voltages, TIME, temperature, and/or Frequency. A table may have one or more dimensions. An expression is a mathematically valid expression, as described in Section 6.

Table 1 – Basic Circuit Elements

Element Name	Description
Resistor	R = <value> (constant, function, or table)
Capacitor	C = <value> (constant, function, or table)
Inductor	L = <value> (constant, function, or table)
Isource (current source)	I = <value> (constant, function, or table)
Vsource (voltage source)	V = <value> (constant, function, or table)
Vcontrolled	V = <value> (function or table)
Icontrolled	I = <value> (function or table)
tline	Single (uncoupled) line: Z0=<value>, Td=<value>, Len=<value> - or - R=<value>, L=<value>, G=<value>, C=<value>, Len=<value> Multi-conductor line: LEN=<value>, RlcgFile=matrix name
Subcircuit	A subcircuit with explicit ports and explicit parameters

Table 2 lists event-detection and other complex elements that are required to support existing IBIS model constructs.

Table 2 – Additional Circuit Elements for Backwards Compatibility

Name	Description
Trigger	Trigger an action based an event or state change
Alarm	Monitor a condition (run time) and report a result
Driver	Complex primitive used to insure backwards compatibility
Reshape	Generates a digital pulse, reshaped from its input ?
Delay	Out = in, delayed by TD
Voltage controlled delay	Out = in, delay is a function of V (future release) ?

Instantiating Circuit Elements

The syntax for instantiating (using) two terminal and controlled source circuit elements in a circuit description is shown below:

Two terminal elements:

element_name instance_name (node_list) symbol=value

Controlled Sources:

element_name instance_name (node_list) (control_node_list) symbol=value

The *element_name* is the name of the element as shown in Table 1 and Table 2 above. The *instance_name* is used to label a particular instance of the element, and must be separated from the *element_name* by one or more whitespaces.

Following the *instance_name* is a *node_list* that specifies which circuit nodes the element is connected to. A comma or whitespace is required between the nodes of a node list, and the parentheses around the list are required. In the case of a controlled source element (icontrolled, vcontrolled) a separate *control_node_list* is also present. The *control_node_list* specifies which circuit nodes are connected to the controlled sources control terminals. As with a *node_list*, a comma or whitespace is required between the nodes of a control node list, and parentheses around the list are required.

Node lists are position sensitive. That is, the first node in the list of an instance is matched to the first terminal in the element primitive, the second is matched to the second, and so forth.

Finally, the element is assigned a value. This assignment is shown as *symbol=value*, where *symbol* is one of the following:

Value is a:	Symbol to use
Resistance	R
Capacitance	C
Inductance	L
Voltage	V
Current	I
Admittance	G
Characteristic Impedance	Z0
Propagation Delay	Td
Length	Len

The equals sign (=) between the symbol and the value is required. Whitespace may be used on either or both sides of the '=' character.. The '=' is followed by a value, which may be constant, a symbolic constant, a table name with an argument list, or an expression. When used, an expression must be enclosed in parentheses. Note that a transmission line element (tline) may have multiple *symbol=value* pairs.

Basic Element Descriptions

Following is a detailed description of the basic circuit elements. As in IBIS, the SI system of units is used (volts/amps/ohms/seconds) for all circuit elements.

resistor

A resistor is a two-terminal circuit element that represents the ratio of voltage across its terminals to the current through the component ($R=V/I$). Units are Ohms. Resistors are instantiated in a circuit description as follows:

```
resistor instance_name (n1 n2) R=value
```

Alternate representations:

```
resistor instance_name (n1 n2) I=value
```

```
resistor instance_name (n1 n2) V=value
```

where:

resistor	Element type
<i>instance_name</i>	The name of this particular instance of resistor
<i>n1 n2</i>	Connection node names
R	Symbol (<i>I is also allowed, see the note below</i>)
<i>value</i>	The value of the resistor element.

A resistor may be assigned a fixed value, the value of a symbolic constant, a value resulting from a table lookup, or the results of an expression. The expression may be a function of time, temperature, frequency or the voltage across the element's terminals.

Inclusion of the following text in red TBD

Note that the resistor element supports using the unit symbol 'T'. In other words, instead of using an expression that specifies the elements resistance, the expression may describe the I/V characteristics of the element as an explicit equation that defines $I=f(v)$. Model makers may use this property to describe a general, two-terminal 'voltage controlled current source' or a two terminal elements such as a diode. See the example below.

Examples:

```
resistor R12 (nodeA nodeB) R=3.3K           | fixed value
resistor my_pullup (pad vcc) R=pullup       | symbolic constant
resistor lkup_r (A B) R=[PD Table] v(A,B)   | table lookup
resistor ramp_r (n5 n6) R=Time*1k          | expression
```

TBD

Examples of explicit I/V specification

```
resistor lookup (pad gnd) I=[PWR Clamp] (v(pad,gnd))
```

```
resistor diode (pad gnd) I=I0(exp(v(pad,gnd)/2*26mV - 1)
```

capacitor

A capacitor is a two terminal circuit element that accumulates charge, where the value of capacitance is a measure of the amount of charge accumulated for a given rate of change of voltage across its terminals ($I = C*d_{dt}(V)$). Units are Farads. Capacitors are instantiated in a circuit description as follows:

capacitor *instance_name* (*n1 n2*) C=*value*

where:

capacitor	Element type
<i>instance_name</i>	The name of this particular instance of capacitor
<i>n1 n2</i>	Connection node names
C	Symbol
<i>value</i>	The value of the capacitor element.

A capacitor may be assigned a fixed value, the value of a symbolic constant, a value resulting from a table lookup, or the results of an expression. The expression may be a function of time, temperature, frequency or the voltage across the element's terminals.

Examples:

capacitor C12 (nodeA nodeB) C=3.3pF		fixed value
capacitor padcap (pad vcc) C=C_comp		symbolic constant
capacitor lkup_C (A B) C=[C Value] v(A,B)		table lookup
capacitor ramp_c (n5 n6) C=Time*1pF		expression

inductor

An inductor component is a two terminal element that accumulates magnetic flux, where the value of inductance is a measure of the amount of magnetic flux per a given rate of change of current through the component ($V = L * d_{dt}(I)$). Units are Henries. Inductors are instantiated in a circuit description as follows:

inductor *instance_name* (*n1 n2*) L=*value*

where:

inductor	Element type
<i>instance_name</i>	The name of this particular instance of inductor
<i>n1 n2</i>	Connection node names
L	Symbol
<i>value</i>	The value of the inductor element.

An inductor may be assigned a fixed value, the value of a symbolic constant, a value resulting from a table lookup, or the results of an expression. The expression may be a function of time, temperature, frequency or the voltage across the element's terminals.

Examples:

inductor C12 (nodeA nodeB) L=3.3nH		fixed value
inductor padl (pad out) L=L_pkg		symbolic constant
inductor lkup_L (A B) L=[L Value] v(A,B)		table lookup
inductor ramp_l (n5 n6) L=Time*1nH		expression

vsorce

A vsorce is a two terminal circuit comp element representing an independent voltage source. The voltage (potential difference) between the terminals of vsorce is referenced from

the first node listed in the node list to the second node listed in the node list as illustrated below. Units are Volts.

```
n1 >--<vsource>--> n2
+   Voltage   -
```

Voltage sources are instantiated in a circuit description as follows:

```
vsource instance_name (n1 n2) V=value
```

where:

<i>vsource</i>	Element type
<i>instance_name</i>	The name of this particular instance of voltage source
<i>n1 n2</i>	Connection node names
V	Symbol
<i>value</i>	The value of the voltage.

A voltage source may be assigned a fixed value, the value of a symbolic constant, a value resulting from a table lookup, or the results of an expression. The expression may be a function of time, temperature or frequency.

Examples:

```
vsource V_123 (n1 n2) V=3.3v           | fixed value
vsource pwr (vcc gnd) V=[Voltage Range] | symbolic constant
```

isource

An isource is a two terminal circuit element representing an independent current source. The positive current is into the first node listed in the node list as illustrated below. Units are Amps.

```
n1 >--<isource>--> n2
Current ---->
```

Current sources are instantiated in a circuit description as follows:

```
isource instance_name (n1 n2) I=value
```

where:

<i>isource</i>	Element type
<i>instance_name</i>	The name of this particular instance of voltage source
<i>n1 n2</i>	Connection node names
I	Symbol
<i>value</i>	The value of the voltage.

A current source may be assigned a fixed value, the value of a symbolic constant, a value resulting from a table lookup, or the results of an expression. The expression may be a function of time, temperature or frequency.

Examples:

```
isource I_123 (n1 n2) I=3.3mA           | fixed value
isource I_source (vcc pad) I=[Current] | symbolic constant
```

vcontrolled

A vcontrolled is a controlled voltage source. The output voltage is a function of the voltage between the controlling nodes as illustrated below. Units are Volts.

```
o1 >--<vcontrolled>--> o2      voltage_in(ctl1, ctl2,...ctlN)
+   voltage_out      -
```

Controlled voltage sources are instantiated in a circuit description as follows:

```
vcontrolled instance_name (o1 o2) (ctl1 ctl2...ctlN) V=value
```

Where:

vcontrolled	Element type
<i>instance_name</i>	The name of this particular instance of controlled voltage source
<i>o1 o2</i>	Output voltage node names. Node 'o1' is positive with respect to 'o2'.
<i>ctl1 ctl2...ctlN</i>	Control voltage node names.
V	Symbol
<i>value</i>	The value of the output voltage.

The control node list is required and must list at least two control nodes; the maximum number is implementation dependent. A controlled voltage source may be assigned the value of an expression or a table lookup. The expression may express a function of temperature, time, frequency and the voltages listed in the control node list. Note that any nodes referenced by the table lookup or expression must be listed in the control node list. Current control is not permitted.

Example:

```
vcontrolled v_123 (n1 n2) (vcc gnd ) V=[V Table] (v(vcc,gnd))
```

icontrolled

An icontrolled is a controlled current source. The output current is a function of the voltage between the controlling nodes as illustrated below. Units are Amps.

```
o1 >--<icontrolled>--> o2  voltage_in(ctl1, ctl2,...ctlN)
Current_out ---->
```

Controlled current sources are instantiated in a circuit description as follows:

```
icontrolled instance_name (o1 o2) (ctl1 ctl2...ctlN) I=value
```

Where:

icontrolled	element type
<i>instance_name</i>	The name of this particular instance of controlled current source
<i>o1 o2</i>	output current node names. Current flows from o1 to o2.
<i>ctl1 ctl2...ctlN</i>	Control voltage node names.
I	Symbol
<i>value</i>	The value of the output current.

The control node list is required and must list at least two control nodes; the maximum number is implementation dependent. A controlled current source may be assigned the value of an expression or a table lookup. The expression may express be a function of temperature, time, frequency and the voltages listed in the control node list. Note that any nodes referenced by the table lookup or expression must be listed in the control node list. Current control is not permitted.

Example:

```
iconrolled gc (pad gnd) (pad gnd) V=[GND Clamp] (v(pad,gnd))
```

The tline Element

A tline is an N-terminal element representing one or more transmission lines. There are three different formats for instantiating transmission line elements – two for single-conductor transmission lines and one for multi-conductor lines.

Single-conductor transmission line:

```
tline instance_name (in1 ref1 out1 ref2) Z0=value, Td=value, Len=value
```

```
tline instance_name (in1 ref1 out1 ref2) Len=value, C=value, L=value, <R=value, G=value>
```

Multi-conductor transmission line:

```
Tline instance_name (in1 in2...inN ref1 out1 out2...outN ref2) RlcgFile=matrix_name
```

where:

tline	element name
instance_name	the name of this particular transmission line instance
in1 ref1	input node(s) (signal and reference)
out2 ref2	output node(s) (signal and reference)
Z0	the transmission lines characteristic impedance
Td	the transmission lines propagation delay per unit length
Len	the transmission lines length, in arbitrary units
C, L, R, G	capacitance/inductance/resistance/admittance per unit length
Value	value, either a fixed value or a symbolic constant
RlcgFile	The name of an RLCG matrix.

Single-conductor tline elements are used to describe a single, uncoupled transmission line. Multi-conductor tline elements are used to describe a set of transmission lines, and the description can include coupling.

The node list of a single-conductor tline element must list exactly four nodes: the input node of the conductor, the input reference node, the output node of the conductor, and the output reference. A multi-conductor tline element can list an arbitrary number of nodes.

<BIG NOTE: The multi-conductor format follows the HSPICE docs. We need to discuss how the ref1 ref2 conductors map into our N-port matrices... do we need N+1 matrices??>

Single-conductor transmission line

A single-conductor lossless transmission line is described using the parameters Z0 ('Z' followed by the number '0'), Td and Len. Z0 is the characteristic impedance. Td is the propagation delay of the line given in delay per unit length, while Len is the physical length of

the line in arbitrary 'units'. Each symbol=value pair is separated by a comma or whitespace, and the pairs can be listed in any order.

Following is an example that describes a five inch long transmission line with a characteristic impedance of 62 ohm and a propagation delay of 200pS per inch.

Example 1:

```
tline example1 (in gnd out gnd) Z0=62, Td=200pS, Len=5
```

If the 200pS/in and 5 inches were converted to sec/meter and meters, the line would be described as follows:

Example 2:

```
tline example2 (in gnd out gnd) Z0=62, Td=7874pS, Len=.127
```

Note that because Td is given in delay per unit length, both descriptions result in the same total delay thru the transmission line.

Note: The 'per unit length' format was chosen to support the existing IBIS .ebd file description.

A single-conductor transmission line can also be described using the physical properties of capacitance, inductance, resistance and admittance per unit length. The advantage of this description over using Z0/Td is that conductor and dielectric losses can be included. The symbol=value pairs for Len, C and L are required while R and G are optional. As before, each symbol=value pair is separated by a comma or whitespace and can be listed in any order.

The following shows the transmission line of example 1 (Z0=62, Td=200pS/inch) described using this format. The ohmic (resistive) loss of the conductor is also included.

Example 3:

```
tline example3 (in gnd out gnd) Len=5, C=3.23pF, L=12.4nH,  
R=0.02
```

Multi-conductor Transmission line

A multi-conductor transmission line is described using an RLCG matrix. This matrix allows the user to fully describe the transmission line segment, including coupling between conductors. The RlcgFile symbol points to an RLCG matrix contained either in the current .ibs file or an external .ibs file.

NEED MORE INFORMATION HERE. I propose we have a keyword that points to a set of matrixes that describe a transmission line section. Just like package info, the matrix info can reside in the current file or in an external file that the EDA tool goes looking for if it can't find the name in the current file.

Complex Elements

In order to support IBIS functionality the following elements are included in the list of primitive elements defined by the IBIS-ML

Trigger

A trigger is an element used to sense the occurrence of an event (usually a state change) then trigger some action. It does this by propagating the time of an event. Triggers are instantiated in a circuit description as follows:

Trigger *trigger_name* (*trigger_expression*)

Where:

trigger element name

trigger_name the name of this trigger element

trigger_expression an expression that evaluates to true or false (i.e. Boolean expression)

The value of a trigger is obtained by evaluating the *trigger_expression*. The parentheses around the trigger expression are required. While the trigger expression is false, the value of a trigger is undefined and any expression involving the trigger should evaluate to zero. When the trigger expression changes from false to true, the trigger assumes the value of the current simulation time (i.e. it becomes defined), and an expression involving the trigger can be evaluated. The trigger value remains fixed at this time until the trigger expression evaluates to false, at which time it reverts back to undefined.

Triggers are generally used to start and/or shift a waveform table. To do this the *trigger_name* is used as part of the expression that forms the index into a waveform table. See the following example:

Example:

```
trigger TR (Logic(control) == 1)
trigger TF (Logic(control) == 0)
.
.
vsource v_up (pad, vcc) V=[Ramp Up] (Time - TR)
vsource v_dn (pad, gnd) V=[Ramp Dn] (Time - TF)
```

In the above example, there are two trigger elements: TR and TF. Trigger TR is activated with the logic node 'control' is equal to the value '1' and trigger TF is activated when the node 'control' is 0.

These triggers are part of the expressions that forms the indexes into the tables [Ramp Up] and [Ramp Dn]. Assume that the node control is initialized to a zero. Because no state change has occurred, the value of trigger is undefined and the expressions (Time - TR) and (Time - TF) are zero. When control is changed to '1', a state change has occurred. The value of TR becomes the current simulation time, and as simulation time is advanced the table [Ramp Up] is processed. When control is changed to a '0', trigger TR goes back to undefined, trigger TF is defined and the table [Ramp Dn] is processed.

Alarm

An alarm is a...

Driver

A Driver element is...

Delay

A delay element is...

Assigning Values To An Element

The value assigned to a element can be expressed in a number of ways. The value may be an explicit value (a constant), a symbolic constant, a value derived from a table lookup, or an expression made up of any or all of the above. This section explains the syntax and details of expressing component values.

Constant

An element value can be a explicit numeric value. A scaling factor and the units, as shown below, can follow the numeric value:

Example:

```
capacitor C12 (nodeA nodeB) C=3.3pF | C=fixed value
```

Symbolic Constant

A symbolic constant is a placeholder for a value that is filled in by the data contained in an IBIS [Model], where the name of the placeholder matches the name of a keyword or sub-parameter in the [Model]. The IBIS-ML parser shall signal an error if a symbolic constant is used in the circuit description but the corresponding sub-parameter or keyword is not present in the [Model].

In the first example, the symbol 'C_comp' is a symbolic constant, and its value is that of the sub-parameter 'C_comp' in the IBIS [Model]. In the second, the symbolic constant matches the IBIS keyword [Voltage Range]. When a keyword is used as a symbolic constant the square brackets are considered part of the name of that keyword.

Example:

```
capacitor C12 (pad gnd) C=C_comp
vsource v_supply (vcc gnd) V=[Voltage Range]
```

Table

Elements can be assigned the result of table lookup, using the syntax shown below.

Symbol= *table_name (index_expression, <variable= index_expression, >...)*

Where

Symbol the unit symbol part of an element instantiation

<i>table_name</i>	the name of the table in the [Model]
<i>variable</i>	selects which table in a multi-dimensional table
<i>index_expression</i>	expression that evaluates into a table lookup value

The *table_name* must match one of the tables in the associated [Model]. Note that the table name includes the square brackets. The *index_expression* is used as an index into the first column of the named table. The entire index expression must be surrounded by parenthesis.

Example 1:

```
Icontrolled gc (pad gnd) (pad gnd) I=[GND Clamp] (v(pad, gnd))
```

In the above example, the voltage between the nodes 'pad' and 'gnd' is used to look up the corresponding value in the table [Gnd Clamp].

A multi-dimensional table is actually a set of tables that share the same table name. As described in section 9.3, each table in the set includes a unique name=value pair that is used to distinguish one table in the set from another. Because there are multiple tables, a multi-dimensional table requires multiple index expressions. As with a single table, the first index expression listed provides the index into each table while subsequent index expressions select which table out of the group of tables to use. Note that the second and subsequent index expressions must assign a value to an explicitly named variable. Each index expression is separated by a comma or whitespace.

Example 2:

```
Icontrolled NMOS (drain source) (drain gate source) I=[FET Curves] (v(drain source),Vgs=v(gate source))
```

Assume that an IBIS [Model] contains a multi-dimensional table [FET Curves], where the name=value pair of 'Vgs' distinguishes one table in the set from another. In the above example, the voltage between the nodes gate and source (v(gate source)) selects which [FET Curves] table to use while the voltage between the nodes drain and source (v(drain source)) is assigned to the variable Vgs. The variable Vgs is used to select the proper table out of the set of [FET Curves] tables.

Expression

While expressions are covered in detail in section 6, there are some rules on expression use that apply specifically to elements.

A legal expression may use state variables (Time, Temperature, Frequency), symbolic or other constants, the component's node voltages, and parameters. Currents and external voltages (nodes or voltages not directly connected to an element) cannot be used in expressions. When used, an expression shall be enclosed in parentheses.

Element expressions may use the following set of operators:

Arithmetic	+ - * /
Trig	sin, cos, tan, arcsin, arcos, arctan
Exponential	exp, exp10, log, log10

One common operation is to specify the voltage between two nodes. The voltage at node1 with respect to the voltage at node2 is expressed as follows:

V(node1 <,| > node2)

The letter 'V' is followed by the two node names. The node names are surrounded by parentheses and are separated by a comma or whitespace.

Following are some examples showing various types of expressions.

```
| Assigning an explicit value
resistor r1 (pad gnd) R=10K

| Assigning a scalar symbolic value
capacitor c_output (output gnd) C=C_comp

| Using a simulator state variable in an expression
resistor r1 (pad gnd) R=(10K*(Temperature-273))

| Using a mathematical operation in an expression
vsource sinwave (in gnd) V=(1*sin(2*PI*Frequency))

| Referencing a voltage
capacitor varcap (n1 n2) C=(10pF * 1/10(V(n1,n2)))

| A controlled source example
vcontrolled vcl (vref gnd) (vcc gnd) V=(V(vcc,gnd)/2)
```

Subcircuits

A subcircuit is a user created circuit that can be instantiated into another, higher level circuit. Subcircuits are created in much the same way as new model_types are created -- by use of the [Define] keyword mechanism. However, in this case the [Define] keyword creates a new subcircuit using the syntax shown below.

```
[Define Subcircuit] subckt_name (node_list) <(parameter_list)>
|
| IBIS-ML statements
|
[End Define Subcircuit] subckt_name
```

Where:

[Define Subcircuit]	keyword indicating that a user defined subcircuit is being created
<i>subckt_name</i>	The name of the subcircuit
<i>node_list</i>	list of connection nodes
<i>parameter_list</i>	optional list of parameters
[End Define Subcircuit]	keyword that terminates a subcircuit definition

The [Define Subcircuit] keyword marks the beginning of a subcircuit definition. The name of the subcircuit is given by *subckt_name*. This is followed by a *node_list* which lists the external connection nodes of the subcircuit. As usual, whitespace or a comma must separate the nodes in the node list, and parentheses are required. The node list is followed by an optional *parameter_list*. The parameter list is surrounded by parenthesis, and each parameter in the list is separated by whitespace or a comma. Subcircuit definitions are terminated by the [End Define Subcircuit] keyword.

IBIS-ML statements are used to create the circuit description.

All variable and parameter names are local to that subcircuit.

If a subcircuit contains a symbolic constant, the symbolic constant can be assigned a value only if it appears in the subcircuit's parameter list – i.e. subcircuits cannot read a [Model] directly.

Instantiating a subcircuit

The syntax for instantiating a subcircuit into a circuit description is similar to instantiating other basic elements.

```
subcircuit subckt_name instance_name (node_list) <(parameter_list)>
```

Where

Subcircuit	element name
<i>subckt_name</i>	the name of the subcircuit
<i>instance_name</i>	name of this particular subcircuit instance
<i>node_list</i>	connection nodes
<i>parameter_list</i>	optional list of parameters

The element name 'subcircuit' indicates this element is a user defined circuit with the name *subckt_name*. These are followed by the *instance_name* and the *node_list*. This is followed by an optional *parameter_list*, which is enclosed in parenthesis. While the node list is positional (the first node in the node list is connected to the first node defined in the subcircuit definition, the second to the second, and so forth) parameters are assigned values using the syntax *parameter_name* = *value*. Each parameter_name=value pair is separated by whitespace or a comma.

The parameter list mechanism allows the user to customize the subcircuit on an instance by instance basis. Since symbolic values are limited to their local model in scope, except for the pre-defined state variables, all parameters must be explicitly passed when they are to be used in a subcircuit.

Section 5

EXPRESSIONS, FUNCTIONS, TABLES, AND MATRICES

5.1. Symbolic Values

An symbolic value is a function that evaluates to a *constant* numeric value that is known at the beginning of simulation. A symbolic value (parameter) must be defined before it is used (for example, in determining a component value). A symbolic value cannot depend on Time, or anything that can change with time. A symbolic value cannot depend on any node voltages or pin currents.

The format for an symbolic value is *define* <label> = <value>. “define” is the sub-parameter that tells the system a constant is being defined. “<label>” is the name for this symbolic value; names must be unique within their scope. The “=” sign is required before <value>.

The value can be a single number, a row of numbers, an expression, or a table lookup. Single numbers follow the same rules as for IBIS 3.2 values, which permits use of engineering and scientific notation. A row of numbers follows the IBIS 3.2 syntax for sub-parameters, with columns in the order in *typ/min/max/<others?>*. An symbolic value is evaluated mathematically once per simulation. A table lookup is performed once per simulation.

Examples of symbolic value use:

```
| a simple value
define Omega = 3
| a row of values
define Param2 = 5.5 5.0 6.0
C_comp 2p 1p 3p
| An symbolic value
define DegreesKelvin = (Temperature + 273)
| table lookup
define R_pullup = [R_vs_T], Temperature
```

Symbolic values can be used in expressions, as long as the symbolic value is defined before it is used.

5.2. Expressions

An IBIS-ML expression may use the following operations: arithmetic (+ - · /), trig (sin, cos, tan, arcsin, arccos, arctan), exponential (exp, exp10, log, log10).

An expression is a mathematical expression that is evaluates to a single value. An expression can depend on Time. An expression should be evaluated at each time step.

An expression is always preceded by an “=” sign and must be surrounded by parentheses.

An expression may take more than one line. The end of the expression is when the total number of left parentheses and right parentheses are equal. If a keyword is encountered before the last right parenthesis, the parser must report an error at the keyword line.

Examples of equations used in determining component values:

```

| simple expression
inductor example3 (1, 2) L=(2.7n + 1.5e-9)
| expression used in argument of a table reference
isource (out_pos out_neg) I = [pullup](V(outpos,outneg))
| expression using symbolic value and controlling node
voltages
iconrolled (out1 out2) (in1 in2 in3)
I = (1e-3*V(in1,in2) + 2e-3*(V(in2,in3)/VCC)

```

5.3. Tables

A table is two or more blocks of data, with each data block being formatted with one column for the first independent variable (such as buffer pad voltage), and three columns for the output values (typ/min/max) (such as clamp current). A table can depend on more than one input, in which case the table has one block of data for each additional input value set.

Examples include the IV and VT tables defined in IBIS 3.2.

5.4. Matrices

Two types of matrices are supported. RLGC matrices are used for a lumped-component model, while s-parameter matrices can be used for both lumped and distributed models. S parameter data can also be obtained from measurement.

RLGC matrices, as defined in the IBIS Connector Specification, may be used for any “lumped” characterization. In particular, RLCG matrices can be used for tline and connector components.

S-parameter matrices are used to represent frequency-dependent model data. An s-parameter matrix can be used for any passive structure, including lumped circuits and distributed components (such as tlines). The matrices must be supplied in Touchstone format, which is available in several field solvers as well as from ATE vector-network analyzers. If the Touchstone file is a separate file, it should be formatted as an IBIS-ML library file, and then included using the [Include Library] keyword.

Coupled inductors can be represented using either RLGC matrix or s-parameter matrix format.

5.5. Functions

A function is an expression that is not contained on the same statement as the component. A function is called by reference. The call to the function shall have the same number of arguments as on the function definition.

Functions will (not) be supported in the first release of the IBIS Macro Language.

=====

Section 6

TEST AND CONTROL CONSTRUCTS

=====

This section of the LRM describes the test and control constructs available in IBIS-ML. These constructs allow the user to control how the circuit description is processed on read-in. For example, the user can tell the EDA tool to check for the existence of specific keywords or sub-parameters in an object's IBIS data set, and/or make decisions based on the values of variables. It is important to emphasize that these test and control statements are evaluated at model read-in or compile time. In that sense these control statements are like the preprocessor directives for a standard programming language such as C.

The following test and control statements are available:

Name	Short Description
assert	display a message based on the results of an expression
define	assign a value to a variable
foreach	create an array of statements then fill them in with data
if, else if, else, endif	test an expression
Inherit	include a previously defined object into the one being defined
node	create an internal node
select, case	select a data set or object configuration based on user input
export	

The syntax and use of the above statements are defined in sections 7.1 through 7.8 below.

6.1. The “assert” Statement

Abstract

The "assert" statement evaluates an expression then displays a message if the expression is false.

Syntax

```
assert ( <expression> ) "message"
```

BNF

```
assert: 'assert' '(' expression ')' "" message ""
```

Description

The assert statement is used to perform checks on the data supplied in the data section of an IBIS data file. The condition is tested at compile time and an error is issued, with the indicated message, if the condition tested evaluates to false. The message part of the assert construct is a text string surrounded by double quotes, and is required. The simulator is not required to continue after a failure, because the data is considered to be defective.

It must be possible to fully evaluate the expression at compile time and the expression must evaluate to a Boolean true or false value. For a check that is evaluated at run time, see "alarm".

Example 1

In this example assert is used to limit the range of numeric values allowed for the symbolic name 'Rload'.

```
[Define Model] load_resistor (pin gnd)
resistor Rload (pin gnd) R=Rload
assert (Rload >= 10) "Rload too small"
assert (Rload <= 100) "Rload too high"
[End Model] load_resistor
```

Example 2

In this example, the string variable 'Enable' may have one of two values: "Active-Low" or "Active-High". All other values are illegal. The assert statement in the 'else' part of the if-else construct tests 'Enable' for the value "Active-High" and tells the EDA tool to display the message "Enable: illegal value" if the expression (Enable == "Active-High") is false.

```
[Define Model] xyz
.
.
if (Enable == "Active-Low")
subckt invert (en 0 enable_pin) inverter
```

```

else
  assert (Enable == "Active-High") "Enable: illegal value"
  subckt buffer (en enable_pin) non-invert
end if
.
.
[End Model] xyz

```

6.2. The “define” Statement

Abstract

The define statement assigns a value to a symbolic name.

Syntax

```
define name = <expression>
```

BNF

```
define : 'define' name '=' expression
```

Description

The define statement assigns a value to a symbol. The assigned value may be a constant or the results of evaluating an expression at compile time. The expression must be fully evaluated at compile time and the define statement must occur before the name being defined is used.

Defines may be used in an expression anywhere a scalar variable or scalar constant may be used.

Example 1

In this example the user assigns a constant value to the symbol “default_z”.

```
define default_z = 50
```

Example 2

Here, the symbol VT is assigned the results of an expression evaluation.

```
define VT = .8617087e-4 * (Temperature+ 273.15) || .026
```

6.3. The "foreach" Statement

Abstract

The "foreach" statement operates across an array of statements. These statements contain variables that are filled in with data from a corresponding 'foreach' data array in the IBIS data file.

Syntax

```
foreach <index> in <pointer to data structure in IBIS data file>
  IBIS-ML statement
.
.
  IBIS-ML statement
end foreach
```

or –

```
foreach <index> in <pointer to data structure in IBIS data file>
  IBIS-ML statement
.
.
  IBIS-ML statement
else
  IBIS-ML statement
.
.
  IBIS-ML statement
end foreach
```

BNF

```
false_part : 'else' '\n'
            statement

foreach : 'foreach' index 'in' key_name '\n'
         statement
         false_part
         'end foreach' '\n'
```

Description

The foreach construct iterates across an array of data, allowing the user to place items from the array into statements. The foreach statement is constructed as follows:

A line beginning with "foreach", followed by a symbolic name used as an index, followed by the word "in", followed by a *key_name* (in the form of an IBIS keyword) that points to an array of data in an IBIS data file. Note that the array of data is assumed to be formatted as shown in the 'foreach' data array description in Section 8.

Any number of IBIS-ML statements. These statements include *substitution symbols*. Substitution symbols are placeholders for the data from the IBIS data file or for the foreach index variable itself.

An optional "else" section, identical to the "else" section of an "if" statement.

The line "end foreach" terminate a "foreach" construct.

The "else" section introduces a section that is used only when the key_name is not present in that object's data section. Note that in the case, the IBIS-ML statements following the 'else' clause may not contain the substitution symbols (e.g. there is no data to substitute).

Substitution symbols are of the general form $\$<symbol>\$$ where $<symbol>$ is either the index variable of the foreach statement or an integer that represents the fields in an array of data. The symbol $\$0\$$ represents the first field in a line of data, $\$1\$$ represents the next field, and so on. The substitution symbols may be used in the middle of a string. Example: Foo $\$1\$$ Bar.

Uses in IBIS 3.2

"Foreach" arrays are used without an else section to define the array of submodels for the "[Add SubModel]" construction.

They are used with an else section for "[Driver Schedule]". The else section is used only when there is no "[Driver Schedule]" keyword.

Example 1

In this example, a model is created that consists of either an array of parallel resistors and subcircuits, or a single capacitor 'C_shunt'.

```
[Define Model] parallel (pin gnd)
foreach iii in [A_Batch_Of_Data]
  resistor R$iii$x (pin a$iii$) R = $1$
  subckt X$iii$x (a$iii$ gnd) $0$
else
  capacitor Cx (pin gnd) C=C_shunt
end foreach
[End Model] parallel
```

The corresponding [Model] data set is:

```
[Model] xyz
Model_type parallel

[A_Batch_Of_Data]
Model_A 120 slow
Model_B 176 fast
[End A_Batch_Of_Data]
C_shunt 100p
```

[End Model] xyz

Applying the foreach construct to the above data set yields:

```
resistor R1x (pin a1) R = 120
subckt X1x (a1 gnd) Model_A
resistor R2x (pin a2) R = 176
subckt X2x (a2 gnd) Model_B
```

Note that because the array [A_Batch_Of_Data] exists, the 'else' part of the foreach statement was not executed. Therefore, the netlist does not include C_shunt.

Example 2

Alternately, the above [Model] data set could be as shown below:

```
[Model] abc (pin gnd)
Model_type parallel
C_shunt 100p
[End Model]
```

This results in an equivalent circuit of:

```
capacitor Cx (pin gnd) 100p
```

Because the [Model] did not include the keyword [A_Batch_of_Data], there is no resistor or subckt.

6.4. The "if" Statement

Abstract

The "if" statement, and its companions "else if", "else" and "end if" are used to choose sections of a macro based on the results of a test.

Syntax

```
if (<expression>)
  IBIS-ML statements
end if
```

or –

```
if (<expression>)
  IBIS-ML statements
else
  IBIS-ML statements
```

```

end if

or –

if (<expression>)
  IBIS-ML statements
else if (<expression>)
  IBIS-ML statements
[else if (<expression>)
  IBIS-ML statements
else
  IBIS-ML statements]
end if

```

BNF

```

false_part : 'else' '\n' net_list
            | 'else if' '(' expression ')' '\n'
              net_list
              false_part
            | /* nothing */

```

```

if : 'if' '(' expression ')' '\n'
     net_list
     false_part
     'end if' '\n'

```

Description

The "if" statement is constructed as follows:

1. A line beginning with "if", followed by an expression in parentheses that evaluates to a Boolean true or false at compile time.
2. A "true" section, consisting any number of statements.
3. Zero or more "else if" sections consisting of:
 - 3a. A line consisting of "else if" and an expression, like statement 1 above.
 - 3b. A set of any number of statements, like statement 2 above.
4. An optional "else" section consisting of:
 - 4a. The line "else".
 - 4b. A set of any number of statements, like statement 2 above.
5. The line "end if" to terminate it.

The selection is done at compile time, along with expression evaluation. Its behavior is similar to the "#if" preprocessor directive in C.

The "select" statement sets up a named scope; the "if" statement does not.

Uses in IBIS 3.2

This is used wherever decisions are made based on parameters supplied.

1. The phase flip relating to the "Enable" and "Polarity" keys.
2. Determining whether power is supplied internally or externally.
3. Determining whether submodels are applied "driving" or "non-driving".

Example

6.5. The "inherit" Statement

----- Abstract

The "inherit" statement includes another object into the object currently being defined.

----- Syntax

inherit <base type>

----- BNF

inherit : 'inherit' literal

----- Description

The "inherit" statement brings in another object as a base. The actual mechanism is a direct text substitution of the statements. If this inherit statement is the first statement in a [Define] block, and this [Define] block has no port list, the port list is also inherited.

Example

First, define a base object.

```
[Define Base] base_resistor (pin gnd)
resistor Rload (pin gnd) R=1k
[End Base] base_resistor
```

Now, use the inherit to include this object into 'driver'.

```

[Define Source] driver
.
.
inherit [Base]base_resistor
vsource Vout (pin gnd) V = [Rise](T-Trise) || [Fall](T-Tfall)
.
.
[End Source] driver

```

This is equivalent to:

```

[Define Source] driver (pin gnd)
.
.
resistor Rload (pin gnd) 1k
vsource Vout (pin gnd) V = [Rise](T-Trise) || [Fall](T-Tfall)
.
.
[End Source] driver

```

6.6. The "node" Statement

Abstract

The "node" statement declares the existence of a new node.

Syntax

```
node <node list>
```

BNF

```
node_list : node_list node
          | /* nothing */
```

```
node : 'node' node_list '\n'
```

Description

The node statement declares new (local) nodes that are not declared elsewhere. It is illegal to declare a node more than once in the same scope. Used 'node' to declare local nodes that are not seen outside, and a node statement is required if a node is not listed in a pin list. The scope of the node is the object being defined.

Example

```
[Define Model] res_cap (pin1 pin2)
node internal
capacitor (pin1 internal) 120p
resistor (internal pin2) 1k
[End Model] res_cap
```

In this case, "internal" is a local node.

6.7. The "select" Control Statement

Abstract

The "select" statement is used to choose between sections of a circuit based on a user specified attribute.

Syntax

```
select (<attribute name>)
  default = <value>
  case (<value>)
    IBIS-ML statement
    .
    .
    IBIS-ML statement
  end case
  .
  .
  [case (<value>)
    IBIS-ML statement
    .
    .
    IBIS-ML statement
  end case]
  .
  .
end select
```

BNF

```
default : 'default' '=' literal '\n'
        | /* nothing */

case :   'case' literal '\n'
        net_list
        'end' 'case' '\n'

case_list : case_list case
```

```

|/* nothing */

select : 'select' '(' env_var ')' '\n'
      default
      case_list
      'end select' '\n'

```

Description

The select statement is used to choose a specific internal configuration or behavior of an object, where the choice is based on the value of a user supplied attribute. This attribute usually represents some external condition or assumption that affects the way an object (or group of objects) behaves. For example, a SPST switch can be either open or closed, and a model for this switch would include data sets that describe the behavior of the switch in both open and closed cases. However, the EDA tool needs to know if the switch is to be modeled in the open or closed position (i.e. which description or data set to use). The select construct is the mechanism by which the EDA tool processing the model selects which data set to use, based on user input.

A "select" statement is constructed as follows:

A line beginning with "select", followed by the name of a user defined attribute, in parentheses. The attribute name must be in the form of a string variable.

An optional "default" line specifying which choice to make when the attribute is not defined or passed in. The default value must be the name one of the data sets associated with that attribute.

At least one "case" section consisting of:

A line beginning with "case", followed by the name, in parenthesis, of one the data set (i.e. a keyword in the IBIS data file) associated with that attribute. The value the case clause is testing also identifies the beginning of a scope, which is connected only for this "case".

A list of circuit components. Control lines are not allowed, except for "assert" and "inherit". Others are considered to be errors.

A line "end case".

The "end select" statement terminates the select construct.

It is important to note that the attribute is NOT part of the data set supplied by the IBIS data set. Rather, the user, usually through the EDA tool interface, supplies the attribute. The switching is done as part of preprocessing, in the same pass that selects data sets.

The value of the attribute passed in must match one of the listed case values. If there is a "default" statement, it is legal to not pass in this attribute. It is an error to specify a value that is not listed.

Note that the "select" statement sets up a named scope; the "if" statement does not.

Uses in IBIS 3.2

This is used to implement the "series switch" model type.

 Example

In the example below a model 'on_or_off' is defined. The model has two signal pins and a control attribute "ctrl". A portion of the circuit is selected by the value of the attribute "ctrl".

```
[Define Model] on_or_off (pin1 pin2)

node t1
resistor Rser (pin1 t1) R = (R_series || short)
select (ctrl)
  case "[Off]"
    capacitor C1 (t1 pin2) C = C_shunt
  case "[On]"
    resistor R1 (t1 pin2) R = R_shunt
    capacitor C1 (t1 pin2) C = C_shunt
end select

[End Model] on_or_off
```

The [Model] data section must have an [Off] section and an [On] section. A scalar "R_series" is optional outside of all [Off] and [On] blocks. The [Off] block must define a value for "C_shunt". The [On] block must define a value for "R_shunt" and "C_shunt".

The two "C_shunt" keys are in different scopes, so the fact that the names are the same is irrelevant.

The [Model] data section in the IBIS data file might look like:

```
[Model] switched_load
  Model_type on_or_off

  R_series 10

  [On]
  R_shunt 100
  C_shunt 50p

  [Off]
  C_shunt 100p

[End Model]
```

This expands differently depending on the value of the argument "ctrl". If "ctrl" has a value of "[On]":

```
subckt switched_load pin1 pin2
Rser (pin1 t1) 10
```

```
R1 (t1 pin2) 100
C1 (t1 pin2) 50p
ends
```

If "ctrl" has a value of "[Off]":

```
subckt switched_load pin1 pin2
Rser (pin1 t1) 10
C1 (t1 pin2) 100p
ends
```

 Section 7

 DATA TYPES

This section of the specification defines the data types available for symbolic names as well as the usage syntax within an IBIS model data section or object description. In brief, symbolic names are grouped into one of the four allowable data types: string variables, scalar numeric variables, single tables (single index tables), and table groups (multiple index). In addition, the following organization structures are available: scopes and "foreach" arrays.

The following sections provide details on these data types and their usage.

7.1. Strings

Abstract

This section describes variables with string values

Description

A string variable is a variable whose value consists of a string of alphanumeric characters. The value must begin with an alpha character. A string variable is used primarily in conditional statements in the macro language. Usually, there are only a few legal values.

Assigning values to string variables (data section usage)

A string variable and its value must be expressed on one line, with the variable and its value separated by at least one white space. An equals (=) character between the variable name and its value is optional. For example,

```
Position = OFF
Polarity non-inverting
```

are both legal ways of assigning a value to a string variable.

A string variable that is not defined is considered to be equal to the empty string.

Using string variables in a [Define] section

String variables are most often used in conditionals and asserts. When comparing or testing the value of a string variable the comparison string is surrounded by double quotes (“”) as shown below.

Example:
if (Position == "OFF")

```

.
.
else
  assert (Position != "ON") "Illegal value for Position"
.
.
end if

```

The if statement compares the value of the string variable 'Position' with the string "OFF". Note the use of the double quotes. Likewise, the assert statement compares the value of Position with the string "ON".

7.2. Scalars

Abstract

This section describes variables with scalar numeric values.

Description

A scalar numeric variable is a variable whose value is a number.

Assigning values to scalar numeric variables (data section usage)

A scalar numeric variable and its value(s) must be expressed on one line, with the variable and its value(s) separated by at least one white space. An equals (=) character between the variable name and its value is optional. For example,

```

[Temperature Range] 25C 0C 100C
R_pullup = 50

```

are both legal ways of assigning a value to a scalar numeric variable.

Note that while a scalar numeric variable can have only one value at any one Time, the user may specify several alternate values to assign to the variable. As shown for the [Temperature Range] variable in the above example, this is done by listing these alternate values in a row on the same line as the variable – i.e. the user constructs an array of data with multiple 'columns' but only one row. Column 0 is the variable name, column 1 is the first variable in the row, column 2 is the next variable in the row, and so on.

Note: *The mechanism for selecting which value to use at run time is TBD*

Referencing in a [Define] section

The use of the name of a scalar in an expression is interpreted as to substitute the value. As an example:

```
[Define Circuit] heater (b1 b2)
resistor R123 (b1 b2) R=[Load]
capacitor C123 (b1 b2) C=C_comp
[End Circuit] heater
```

```
[Circuit] xyz
Circuit_type heater
[Load] 50
C_comp 100u
[End Circuit]
```

In this example, the value of R123 is 50 ohms. C123 is 100 microfarads.

7.3. Tables

Abstract

This section describes numeric key-value tables.

Description

A numeric key-value table is a two dimensional array of data consisting of at least two rows and two columns. For each row of data, the independent variable is placed in the left most column (column 0) while the dependent variable(s) are placed in columns 1, 2 etc. The data in column 0 is the *index* for the data in that row. Tables are generally used to define nonlinear transfer functions and points on a waveform.

Constructing tables (data section usage)

Tables begin with a header identifying the table. This header is in the form of an IBIS keyword; i.e. [`<table name>`]. The square brackets surrounding the name are required.

On the line below the header the user may optionally specify the table's *behavior attributes*. There are three behavior attributes; Order, Below and Above.

The construct Order = `<integer>` specifies how the EDA tool should interpolate between points in a data column. Legal values for Order are 0, 1, 2 or 3. 1 means use linear interpolation, 2 means use quadratic spline interpolation, while 3 means use cubic spline. If Order is omitted the EDA tool is free to choose the interpolation algorithm.

The construct Below = `<integer>` specifies how the EDA tool should extrapolate new data points that lie beyond the most negative value in the data column. Legal values for Below are 0, 1, and 2. The value for Below must be less than or equal to Order. A value of 0 means extend horizontally at the boundary value, 1 means to extend the slope as a straight line, and 2 means to extend the slope and second derivative. If Below is omitted, the simulator is free to choose the interpolation algorithm.

The construct `Above = <integer>` specifies how the EDA tool should extrapolate beyond the most positive value in the data column. Values and rules are the same as for the behavior attribute `Below`.

Following the behavior attributes the user may optionally specify additional `<Name>=<value>` pairs. These additional attributes can be used to group tables into the equivalent of multi-dimensional tables. The attributes are interpreted either as an additional index for multidimensional tables, or as data in a scope.

Following the optional behavior and other attributes, the user may optionally place column headings for the following table of numbers. If used, column headings are of the form `<name>` where name is an alphanumeric string. As with the data itself, each column heading must be separated by white space.

Following the attributes and headers is the data itself. This is a list of numbers, organized by row and column. If column headings are supplied, the number of columns must match the number of headings. The number of columns must be consistent within a table. The value "NA" indicates that a data point is omitted, but any column must have at least 2 non-NA values.

A table is terminated by the occurrence of the next keyword – i.e. a line beginning with '[' (left square bracket). Any attributes, lines of data that are mis-formed (do not have the proper number of columns), or other constructs that occur between the last line of the table and the first keyword are ignored.

Note: For consistency, it is recommended that the user terminate a table with a keyword of the form [End <table name>].

Except for multi-dimensional tables (next section), there must be only one table with a particular name.

Following is an example of a table that includes all the optional attributes:

```
[Lookup Table 1]
Order = 1
Below = 1
Above = 1
Resistance = 50
Volts I(typ) I(min) I(max)
0    0    0    0
1    1m   0.9m  1.1m
2    2m   1.8m  2.2m
3    3m   2.7m  3.3m
[End Lookup Table 1]
```

Here is the same data, placed in a table that uses only the required parts of a table.

```
[Lookup Table 1]
0    0    0    0
1    1m   0.9m  1.1m
2    2m   1.8m  2.2m
3    3m   2.7m  3.3m
```

Referencing a table in a [Define] section

To reference a table in an expression, reference it by name, followed by the lookup value in parentheses.

Example:

```
[Define Resistor] non-linear (a1 a2)
resistor Rx (a1 a2) I = [IVcurve](V(a1,a2))
[End Resistor] non-linear
```

The table 'IVcurve' gives the value of resistor 'Rx' where the value of the argument 'V' (in parentheses) is used as an index into column zero of that table.

The corresponding data section for the above [Define] would be as follows:

```
[Resistor] R123
Resistor_type non-linear

[IVcurve]
Volts I(typ)
-5 -0.1
-2 -0.05
0 0.0
2 0.05
5 0.1

[End Resistor]
```

Here this resistor's I-V characteristics are defined by the table [IVcurve].

7.4. Multi-dimensional tables

Abstract

This section describes making multi-dimensional tables.

Description

Multi-dimensional tables are a collection of single tables that use name=value pairs at the beginning of each table to create the additional dimension. Multi-dimensional tables are used to describe a value that might be both nonlinear and time dependent. They are also used to describe multi-terminal components where the dependent variable is nonlinear in relationship to both ports.

Constructing Multi-dimensional tables (data section usage)

Multi-dimensional tables are defined as a group of numeric key-value tables that have the same table name. Multi-dimensional tables are constructed using the same rules as the single dimensional tables, except that all tables in the group are required to have at least one name=value pair in common.

Following is an example of how the multi-dimensional table [IVcurve] is created. Note that each table includes the name-value pair Temperature=<value>.

```
[IVcurve]
Temp = 25
Volts I(typ)
-5 -0.1
-2 -0.05
0 0.0
2 0.05
5 0.1
```

```
[IVcurve]
Temp = 0
Volts I(typ)
-5 -0.2
-2 -0.05
0 0.0
2 0.05
5 0.2
```

```
[IVcurve]
Temp = 100
Volts I(typ)
-5 -0.3
-2 -0.05
0 0.0
2 0.05
5 0.3
```

IBIS 3.2 compatibility

Multi-dimensional tables are used for the following keywords in the [Model] section:

* Waveform
Series MOSFET

Referencing a multi-dimensional table in a [Define] section

As with single numeric key-value tables, multi-dimensional tables are referenced by name, followed by a two or more lookup values, separated by commas, in parentheses.

Example:

```
[Define Resistor] non-linear (a1 a2)
resistor Rx (a1 a2) I = [IVcurve](V(a1,a2), Temperature)
[End Resistor] non-linear
```

The first (unnamed) argument in parentheses is used as an index into column 0 of one of the table in the group. The named argument (i.e. Temperature=T) is used to select which table of the group the first argument indexes into.

Instead of the named attribute, a "*" may be used to indicate that multiple tables with the same name and different attributes are allowed, and that it is handled as a special case. An example of this usage is the driver waveform tables found in IBIS 3.2.

7.5. Scopes

Abstract

A scope is a named set of data delimited by keywords in brackets.

Description

When constructing an IBIS data section, a portion of the data can be set aside in a separate scope. A scope is a set of data that can be referred to and operated on by name. This is useful for grouping set of data so they can be operated on by a "select" or other macro language statement. Also, additional attributes in a table are interpreted as being in a scope.

Constructing scopes (data section usage)

A scope is constructed as follows:

A scope begins with a scope header in the form of a traditional IBIS keyword in square brackets; i.e. [`<scope name>`]

Note: While not required, it is recommended that the <scope name> begin with the literal 'Begin'.

The body of a scope contains other symbols, including string variables, scalars, tables, or other scopes. Scopes may be nested.

Any line that is not a proper member of the scope terminates the scope. Specifically, the occurrence of the next keyword terminates a scope.

Note: While not required, it is recommended that a scope be terminated with a keyword of the form
[End `<scope name>`]

The attributes (behavior attributes or name=value pairs) in a table are accessible as if the table were a scope.

A case statement in the macro language, if selected, brings a scope with a matching name into the enclosing scope.

 IBIS 3.2 compatibility

Scopes are used in the basic sense for [Model_Spec] and [Submodel_Spec]. Select and case statements, and the corresponding use of scopes, are used for the selection associated with the Series_switch model_type. Attributes of a table being treated as a scope are used to describe a test load circuit in [Falling_Waveform] and [Rising_Waveform].

 Referencing a scope in a [Define] section.

The macro languages access data in a scope by prefixing the data with the scope name. For example, a reference to “[Model_Spec] Vmeas” in the macro language accesses the variable Vmeas in the [Model_Spec] scope.

Scopes are also used in the ‘case’ portions of a select statement. In the example below, the user defines the value of “circuit_selector” as either [Lumped] or [Distributed]. In the data section the user then creates the corresponding data scopes [Lumped] and [Distributed]. Note that because the data section does not contain any other named scopes, any other value of circuit_selector is illegal.

```
[Define Test] of_select (pin gnd pad gnd)
select (circuit_selector)
  case "[Lumped]"
    capacitor C1 (pin gnd) C = C_pin
    inductor L1 (pin pad) L = L_pin_pad
    capacitor C2 (pad gnd) C = C_pad
  end case
  case "[Distributed]"
tline T1 (pin gnd pad gnd) delay = Delay z0 = impedance
  end case
end select
[End Test] of_select
```

```
[Test] TL_test
Test_type of_select
[Lumped]
  C_pin = 1.2p
  L_pin_pad = 374p
  C_pad = 1.24p
[End Lumped]
[Distributed]
  Delay = 2.5n
impedance = 53
[End Distributed]
[End Test]
```

7.6. Foreach arrays

Abstract

A set of data can be described using "foreach" arrays.

Description

A 'foreach' array is a named set of macro language statements that are operated on by a foreach construct. The macro language statements are usually some type of assignment statement with an arbitrary number of fields that are filled in by the foreach operator.

Constructing foreach arrays (data section usage)

A foreach array is constructed as follows:

A foreach array begins with header that identifies the array. This header is in the form of a traditional IBIS keyword; i.e. [<foreach array name>]

The body of a foreach array consists of an arbitrary number of lines with an arbitrary number of fields in each line. The fields are delimited by whitespace, and can contain no whitespace within them.

A foreach array is terminated by the occurrence of the next keyword; i.e. line that begins with a left square bracket '['.

Referencing a foreach data array in a [Define] section

As described in section 7, the data in a foreach array is referenced using a "foreach" statement:

```
foreach foo in [A_Batch_Of_Data]
```

The loop variable (foo in this example) is assigned a sequential number (1,2, ...) which can be accessed inside a macro language statement as \$foo\$. The fields on each line are accessed by number: \$0\$, \$1\$, etc. So, for the foreach array [A Batch Of Data]:

```
[A_Batch_Of_Data]
Model_A 120 slow
Model_B 176 fast
[End A_Batch_Of_Data]
```

The macro language might describe:

```
foreach iii in [A_Batch_Of_Data]
resistor R$iii$x (pin a$iii$) R = $1$
subckt X$iii$x (a$iii$ gnd) $0$
end foreach
```

This combination is equivalent to:

```
resistor R1x (pin a1) R = 120
subckt  X1x (a1 gnd) Model_A
resistor R2x (pin a2) R = 176
subckt  X2x (a2 gnd) Model_B
```

A foreach statement may have an "else" section which is executed when the foreach array does not exist in the data section.

=====
=====
Section 8FUTURE EXTENSIONS
=====
=====

This section of the LRM describes extensions that reached a consensus within the IBIS-ML committee that these items are beyond the scope of the first release. Although these are not be part of the first release of the IBIS-ML specification, these extensions are expected to be included in a future release, with the syntax TBD.

This section is included here to provide information to both users and EDA vendors on anticipated extensions. There is no guarantee that these extensions will be adopted as presented here.

Expressions will be expanded to include Frequency as an allowed state variable. This will allow support of frequency-domain models and analysis.

Math operators will be expanded to include derivatives (d_dt) and integrals (integral_t) with respect to Time.

Components will be allowed to depend on terminal currents as well as node voltages. Terminals must be passed in as controlling values, in a format TBD:

*controlled_component_name instance_label (node list) (control node list)(control pin list
symbol = expression(voltages on nodes, currents through pins)*

=====
 =====
 temporary use only
 =====
 =====

This is stuff that should be removed from the IBIS-ML spec but not lost.

Background:

Current EDA tools developed specifically for signal integrity simulation and analysis apply proprietary simulation algorithms to a set of well defined and understood data, such as that contained in IBIS files. The data model, in turn, is based on a standard I/O buffer model topology. In other words, it was assumed that the I/O buffers used for digital logic can be abstracted to a set of controlled voltage and current sources connected in a standard manner, where components can be removed but not added. Given this fixed model, the user needs only to supply specific data for each I/O buffer. This data consists primarily of the DC I/V (output voltage vs. current) characteristics, plus V/T (output voltage vs. time) waveforms that describe an output's transition from one state to another. In addition, other "data book" type information is included in the format specification, to allow inclusion of information needed for system level timing analysis and error checking.

The original IBIS specification was the first industry standard method for electronically transporting I/O buffer modeling data. However, while the original IBIS specification has obtained wide industry acceptance, it has become increasingly clear that the original paradigm of data applied to a *fixed* model is no longer able to fully meet the needs of the industry. This is due to a number of issues, some of which are:

- The increasing complexity of I/O buffers
- The increasing use of programmable I/I buffers
- The advent of new signaling technologies
- Simulation methodologies that require describing the input-to-output relationships of a digital receiver, and finally
- The need to accurately model the effects of on-die power distribution, return paths, and pin to pin coupling on an I/O pin's behavior.

Therefore, the IBIS-ML macro language is proposed.

Relationship between IBIS and IBIS-ML:

One of the fundamental concepts behind IBIS is the separation of data and algorithm. I/O buffer modeling data is contained in an IBIS file proper, while (as described above) the algorithm(s) that operate on the data are encoded in a EDA tool and cannot be changed by the user.

Now, with the definition of IBIS-ML, this separation of data and algorithm becomes explicit. IBIS data files contain model data – simulation data as well as ‘data book’ specifications – while the corresponding IBIS-ML file defines the behavior and usage of models (and other objects) that are referenced in the IBIS-ML file. Note that both types of files are still considered ‘IBIS’ files and both use the ‘.ibs’ extension.

Commitment to Backwards Compatibility:

Support for IBIS versions is provided through the IBIS Standard Library.