CMC Standard Netlist Language and Model File Format

Revision History

Release	Date	Authors	Comments
-1.0.0	10/15/2008	S. Mertens	DRAFT: assembling requirements
-1.0.1	11/12/2008	S. Mertens	Adding examples and small changes after phone conference
-1.0.2	11/25/2008	M. Kole and S. Mertens	Adding more examples for Verilog-A and redefining requirements after comments from phone meeting
-1.0.3	12/09/2008	M. Kole and S. Mertens	Minor changes to formatting and a few more examples and redefining requirements after comments from phone meeting.
-1.0.4	12/16/2008	S. Mertens	Minor changes after feedback from face-to-face meeting at CMC
-1.0.5	1/14/2009	S. Mertens	Minor changes after phone meeting
-1.0.6	3/16/2009	S. Mertens	Minor changes after phone meeting
-1.0.7	3/30/2009	S. Mertens	Removed ADS examples after Q1 meeting
-1.0.8	4/7/2009	S. Mertens	Changes after phone meeting
-1.0.9	4/23/2009	S. Mertens	Changes after phone meeting
-1.0.10	4/29/2009	S. Mertens	Adding examples
-1.0.11	5/8/2009	S. Mertens	Added examples from Madan Nuttaki and comments from meeting
-1.0.12	6/1/2009	S. Mertens	Added example
-1.0.13	6/3/2009	M.Kole	Added Verilog-A examples

Release	Date	Authors	Comments
-1.0.14	9/28/2009	S. Mertens	Removed Cadence-Spectre examples and replaced with Berkeley-Spectre
-1.0.15	10/1/2009	S. Mertens	Added examples using ADS syntax with help from R. Poore
-1.0.16	10/7/2009	S. Mertens	Removed inline and added example with multi-line statement after phone meeting
-1.0.17	11/18/2009	S. Mertens	Description of the language based on Berkeley-Spectre/ADS
-1.0.18	12/14/2009	S. Mertens	Made changes after face-to-face meeting
-1.0.19	1/13/2010	S. Mertens	Made changes after phone meeting
-1.0.20	2/5/2010	S. Mertens	Made changes after phone meeting
-1.0.21	2/25/2010	S. Mertens	Made changes after phone meeting
-1.0.22	3/12/2010	S. Mertens	Made changes after phone meeting
-1.0.23	4/7/2010	S. Mertens	Made changes after phone meeting
-1.0.24	4/21/2010	S. Mertens	Made changes after phone meeting
-1.0.25	5/14/2010	S. Mertens	Made changes after phone meeting
-1.0.26	6/4/2010	S. Mertens	Made changes after phone meeting
-1.0.27	7/7/2010	S. Mertens	Made changes after phone meeting
-1.0.28	8/5/2010	S. Mertens	Made changes after phone meeting
-1.0.29	10/8/2010	S. Mertens	Made changes after phone meeting
-1.0.30	11/23/2010	S. Mertens	Made changes after phone meeting
-1.0.31	12/1/2010	S. Mertens	Preparing for evaluation
-1.1.0	12/13/2010	S. Mertens and B. Peddenpohl	Fixing typos for evaluation
-1.1.1	3/15/2011	R. Poore and S. Mertens	Editorial changes to improve clarity
-1.1.2	3/17/2011	R. Poore and S. Mertens	Changes from Agilent feedback approved by committee
-1.1.3	4/13/2011	S. Mertens	Changes from Synopsys and Simucad feedback and addition of more formal BNF formulation
-1.1.4	4/27/2011	S. Mertens	Made changes after phone meeting
-1.1.5	5/4/2011	S. Mertens	Made changes after phone meeting and preparation for evaluation
-1.1.6	5/11/2011	S. Mertens	Changed name of mextram according to wishes of Prof. van der Toorn
-1.1.7	8/25/2011	S. Mertens	Made changes to resolve negative comments from first vote after phone meeting
-1.1.8	9/15/2011	S. Mertens	More changes to resolve negative technical comments
-1.1.9	9/29/2011	S. Mertens	Still resolving negative technical comments – preparing for final draft for evaluation
-1.1.10	10/26/2011	S. Mertens	Preparing for final draft for evaluation
-1.1.11	11/2/2011	S. Mertens and M. Nutakki	Preparing for final draft for evaluation
-1.1.12	11/17/2011	S. Mertens	Preparing for final draft for evaluation

Release	Date	Authors	Comments
-1.1.13	11/17/2011	S. Mertens	Reviewing draft for evaluation
-1.1.14	11/17/2011	S. Mertens	Finalizing draft for review
-1.1.15	11/17/2011	S. Mertens	Fixed some small issues for next evaluation
0.0.0	4/5/2012	S. Mertens	First approved standard

Review History

Revision	Date	Reviewers

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

Samuel Mertens, Agilent, *chair*

Geoffrey Coram, Analog Devices, Inc.
Ryan Eatmon, Texas Instruments
Pascale Francis, National Semiconductor
Yoshiharu Furui, Silvaco
Ahmed Gamaleldin, Mentor Graphics
Ben Gu, Freescale
Steven Hamm, Freescale
Marq Kole, NXP
Colin McAndrew, Freescale
Shahriar Moinian, LSI
Madan Nutakki, IBM
Bob Peddenpohl, Cypress
Rick Poore, Agilent
Ahmed Ramadan, Mentor Graphics
Saibal Saha, Cadence
Haruyuki Taniguchi, Sony
Jushan Xie, Cadence
Sheldon Zhang, Synopsys
David Zweidinger, Texas Instruments

Table of Contents

Contents

Contents

1 Introduction

The CMC has set up a subcommittee to work on defining a standard netlist and model file language. This document contains the requirements for the language and is a working draft for the language description. The language has been inspired by the Berkeley Spectre language which was written by Ken Kundert [2] and by the ADS language [3].

2 Requirements

General requirements:

1. The language MUST be defined by a formal grammar.
2. The language MUST have unambiguous definitions of a model card, an instance of a model, a subcircuit, and a library.
3. The language MUST be case-sensitive. The standard SHOULD specify a warning mode, which would warn users when names are being redefined which only vary in their capitalization.
4. The language MUST define the format for real numbers that is to be used.
5. The language MUST understand scale factors (defined in Appendix 1). Unit names and symbols MUST NOT be allowed to be a part of the instance/model card.
6. The language SHOULD be able to use international characters sets. The language MUST define the allowed ASCII characters that may be used in names, and provide an escaping mechanism for specifying other characters.
7. A mechanism to define global nodes MUST be part of the language. The language SHOULD specify how these nodes are defined within a hierarchical structure.
8. The language MUST define legal characters and name spaces, including any length limitations. Reserved keywords that can't be used within their namespace for variable/instance/parameter/node names must be defined.
9. The characters that start a comment, both for a full line and for the remainder of the line MUST be defined in addition to the characters that are used to continue multi-line statements, white space characters, tab spacing , embedded comments, single line comments, block comments and extra spacing. The language SHOULD allow a minimum of such characters.
10. Basic models (R, L, C, mutual inductors, independent sources and controlled sources) and Verilog-A, API generated, or other built-in models MUST be instantiated in a standardized way (Appendix 2).
11. The language MUST NOT contain positional arguments (except the terminal list may be position dependent).
12. The terminal list for a component MUST have a default order. This order MUST be able to be overridden by a specific connectivity directive (recommendation: use the Verilog-A standard). A warning/parsing mechanism has to be defined if the user does not connect the required nodes. A mechanism SHOULD be defined to allow the user to access internal nodes for CMC standardized models.
13. The terminal list for a component MUST (optionally) be defined by delimiters (recommendation: use parentheses "()"). The committee will decide if this is optional or not after exploring examples.
14. A key letter MUST NOT be used to identify a component type. Full names (not level numbers) must be used (Appendix 2). If a model uses a parameter named LEVEL to distinguish between versions or model features, it is to be treated as a model parameter, not as an identifying parameter to distinguish it from another model. If a new version of the model is different enough from a previous model, the model developer will have the option to give it a different name to identify it.
15. The language MUST be able to provide one model card as a subset of another one, including remapping of the model parameters to new names.
16. The model file format SHOULD have similar functionality as the Verilog-A paramset semantics - a lot of our technical requirements are actually already satisfied by this format, such as chaining, automatic model selection (by overloading).
17. The language SHOULD allow conditional instantiation, also with scoping.
18. The model file format SHOULD allow multiple model sets to be used next to each other, for example allow the support of BICMOS and carrier laminate technologies in one simulation environment. Different technology files SHOULD have a different namespace. The language SHOULD have a mechanism to allow the user to switch between these namespaces when generating a netlist. The language needs to clarify what happens when two model sets both define a variable called "foobar" at the top level, or both want to set a global option or if more than one technology file is loaded which contains a model with an identical name.
19. The language MUST allow a model to be defined within a subcircuit, and SHOULD allow to reference this model from another subcircuit. The language SHOULD NOT allow a subcircuit to be defined within another subcircuit, unless more evidence and examples can demonstrate the need for this.

Limitations to the Standard in the first phase

20. The standard will NOT define analysis specific parameters for sources.

Parameters, functions, referencing, hierarchy and scoping requirements:

21. The language MUST support the notion of hierarchy.
22. The language MUST support the notion of scope. It MUST appropriately propagate variables defined at a higher hierarchical level to lower hierarchical levels ("global" variables MUST be included in this). This is required for multi-technology simulation.
23. The language MUST define netlist parameters, netlist functions and expressions. It MUST define the allowable data types and the allowed operators (+ - * /,…) and functions (log10,ln, sin, …) MUST be defined.
24. The language MUST define the syntax and semantics of to refer to instances, models, parameters, nodes and branches, including references at any level of hierarchy, in and out of the subcircuit definition. The language MUST support a method to alias part of the hierarchy.
25. The language MUST define how to pass currents and voltages to all instances and subcircuits where this is needed (eg Verilog-A). The currents could be defined by a probe or a source.
26. The language MUST be able to handle both global statistical variables, common across all components affected by a specific process parameter, and mismatch statistical variables that affect individual instances of components.
27. The language MUST allow instance parameters to be evaluated inside of models. Especially, the multiplicity factor MUST be explicitly available, it is required for proper specification of mismatch variation, and MUST be handled hierarchically. To clarify, the multiplicity factor MUST be accessible and usable in expressions, where the simulator properly expands any hierarchical definitions of multiplicity. For models where the implementation of the multiplicity factor has caused confusion, the standard will provide a specification.
28. The language MUST be able to define statistical relations between parameters in specific statistical parameter sets. The language MUST NOT define how the analysis is implemented.

Interaction with simulator/environment requirements:

29. The language MUST contain syntax to specify a limited set of simulator options, for example "tnom"," temperature" or"scale." These MUST be able to be specified hierarchically. The language SHOULD specify how these simulator options can be set or accessed within different technology files. The set MUST be defined within the standard (Appendix 3). Where possible the name of such parameter SHOULD correspond with its Verilog-A equivalent.
30. Encryption directives SHOULD be defined.
31. The language SHOULD have a structure that allows easy integration in source code documentation tools such as Doxygen.
32. The language SHOULD allow for a preprocessor to handle macros and conditionals. This would mean we have to define preprocessor functionality. The language SHOULD not include scripting, although a mechanism MAY be defined for access to names and values supplied from an exterior shell.
33. The language SHOULD allow parameters that exist only to help convergence or other simulator-specific behavior to be labeled as such so a simulator would be able to recognize, or ignore, them.
34. The language SHOULD be identifiable as such, but not by a particular file name extension. This could be achieved through simulator lang=cmc_standard directives.
35. The language SHOULD support the ability to do range checking of parameters and bias dependent parameters, and issue warnings and errors.

3 Comments/Items to be resolved:

4 Language description

The Backus-Naur form will be used to describe the language. We have used the following conventions

- Boldfaced words and boldfaced symbols enclosed by double quotes are literal keywords that are required in the syntax. If a double quote **"** is a required part of the syntax it will be written as **""""**
- Everything enclosed by square brackets […], which have not been enclosed by double quotes and boldfaced, form an optional part of a statement. When square brackets are a literal part of the syntax they will be bolded and enclosed by double quotes, such as **"["** and **"]"**.

- Parts of any statement which need to be replaced by the user will be enclosed by the smaller than and greater than brackets <…>, which have not been boldfaced and enclosed by double quotes, such as **"<"** and **">"**.
- A vertical bar (|) which is not boldfaced and enclosed by double quotes is used to point to separate alternatives. If a vertical bar is a literal part of the syntax, it will be boldfaced and enclosed by double quotes such as **"|"**.
- Curly braces { }, which are not boldfaced and enclosed by double quotes, are used to identify a repeated item. A repeated item may appear as many times as the user wants, or may be omitted. The repetition occurs to the right of the previous element. If curly braces are a part of the literal syntax they will be boldfaced and enclosed by double quotes **"{"** and **"}"**.
- Some non-printable characters are part of the literal syntax, they are the new line character **\n** and the tab character **\t**. They will be boldfaced to show that they are a literal part.
- Some terms are defined in Appendix 4, those terms are written in italics, when encountered for the first time in this section.
- Examples are enclosed by a box.

4.1 Language rules and definitions

The language is used to describe the topology and characteristics of sets of *instances* in a text file which can be parsed by a simulation tool, the *netlist*.

4.1.1 Statements

Statements are separated by the endline character ("\n"). A statement can be extended over multiple lines by starting the next line with the character "+". The "+" has to be the first character which is not an endline character or a delimiter after the endline character for the line to continue. Blank line(s) may appear within continuation lines and are ignored; the continuation behaves as if the blank line(s) were not present.

//From A7.1

statement::= <statement_line> **\n**
{ **"+"** <statement_line> **\n** | **\n** | <comment> **\n**}
statement_line::= {<printable_ASCII_character> | **\t** }
printable_ASCII_character ::= ASCII : 0x20-0x7E

The printable_ASCII_character is any character which can be printed, decimal number 32 to 126 in the ASCII alphabet.

In the next two examples, the lines on the left hand side are equivalent to those on the right hand side

`R1 (node1 0) resistor r=50`	`R1 (node1 0) resistor` `+r=50`

`subckt someThing (node1 node2)` `parameters`	`subckt someThing (node1 node2)` `parameters arg1=x arg2=y`

`+ arg1=x`
 	
`+ arg2=y`	

4.1.2 Comments

Comments and blank lines are ignored by the parser.

One-line comments are preceded by the characters "//": all text after // until the endline character shall be treated as a comment.
All text between "/*" and "*/" will be considered a multi-line comment.

//From A7.1

comment ::= one_line_comment | multi_line_comment
one_line_comment ::= **"/" "/"**{ <printable_ASCII_character> } **\n**
multi_line_comment ::= **"/" "*"** { <multi_line_text> } **"*""/"**
multi_line_text ::= <printable_ASCII_character> | **\n**

Comment lines and blank lines shall be removed (or ignored) by the parser when parsing the netlist.

The continuation does not continue the comment line; single line comment lines are just that, they comment out one line without interrupting the continuation.

In the following example, the text on the left hand side is equivalent to the one on the right

`//this is a comment`
`inst1 (node node2) mysubcircuit arg2=30`
`// is also a comment`
`/* this is`
` a multiline`
` comment */`
	`inst1 (node node2) mysubcircuit arg2=30`

`R1 (node1 0) resistor`
`+r=50`
`//+ tc1=1.2e-4`
`+tc2=0.023`
`+//dtemp=10`
`+m=2`
	`R1 (node1 0) resistor r=50 tc2=0.023 m=2`

4.1.3 Delimiters (field separators)

A delimiter (field separator) may consist of one or more blanks, tabs, commas or comments. Multiple delimiters will be treated as one by the parser, except for multiple commas or if they are a part of a *string*.

//From A7.1

delimiter ::= { **" "** | **\t** | **","** }

4.1.4 Case sensitivity

The CMC language is case sensitive. Keywords with the exception of scale units on numbers and predefined functions are supposed to be lower case. The parser may provide an optional warning if two variables are only different in capitalization.

In the following example Node2 and node2 will not be viewed as the same terminal by the parser

```
R1 (node1 node2) resistor r=10
R2 (Node2 ground) resistor r=10
```

And the following example shall be flagged for a warning by the parser.

```
variable1=1
Variable1=2
```

4.1.5 Strings

Strings are sets of characters on a single line which are contained between two double quotes ". The allowed characters are those in the ASCII set. Delimiters, comments and other characters lose their special function when they are part of a string. The following characters can be placed inside a string by using an escape character: new line (\n), tab (\t), \ (\\), " (\") and any character by its ASCII code as \ooo (where o is an octal number between 0 and 7). When the ASCII-code \000 is found, the string shall be terminated after the previous character.

//From A7.2

string ::= **"<**text_of_string>**"**
text_of_string ::= {<character>}
character ::= <printable_ASCII_character> | <ASCII_code> | **\n** | **\t** | **"\" "\"** | **"\" """**
ASCII_code ::= **"\"**<octal_number><octal_number><octal_number>
octal_number ::= 0|1|2|3|4|5|6|7

To select a set of characters in a string the following function can be used:

//From A7.2.1

select_string ::= <string_variable_name> **"(**" <i> **":**" <j> **")"**
string_variable_name := <name>
i ::= <integer_number>
j ::= <integer_number>

---

returns a string containing the i-th to j-th character in the string, the first character is numbered 0. If i is less than zero, or j is less than i, or j is greater than the length of the string, then an empty string shall be returned. The string_variable_name is the name of a variable of the string data-type.

4.1.6 Real numbers

Real numbers are described by IEEE std 754-1985. Three notations are allowed: decimal (s#.#), scientific (s#.#es#) and scaled (s#.#scale character), where s is an optional sign character of either "+" or "–" and # is one or more decimal digits.

//From A7.2.2

real_number ::= [<sign>]<unsigned_number>.<unsigned_number> |
[<sign>]<unsigned_number>[.<unsigned_number>]<exponential>[<sign>]<unsigned_number> |
[<sign>]<unsigned_number>[.<unsigned_number>]<scalefactor>
sign ::= **"+"** | **"-"**
unsigned_number ::= <decimal_number>{<decimal_number>}
decimal_number ::= 0|1|2|3|4|5|6|7|8|9
exponential ::= **e** | **E**
scalefactor ::= **a | f | p | n | u | m | k | K | M | G | T**

Numbers cannot contain any spaces. The scale factors are listed below and in Appendix 1. Note that these are the same scale factors defined by Verilog-A.

Units Prefix	Multiplier
a	10^{-18}
f	10^{-15}
p	10^{-12}
n	10^{-9}
u	10^{-6}
m	10^{-3}
K, k	10^{3}
M	10^{6}
G	10^{9}
T	10^{12}

Only one scale factor can be used in a number. Just like Verilog-A, scale factors can't be used in combination with the scientific notation. When a number is defined using the scientific notation any scale unit will be ignored. All text after a scale factor will be ignored by the parser until the next delimiter.

The following two lines have the same meaning:

```
C1 (node1 0) capacitor C=1u
```

```
C1 (node1 0) capacitor C=1uF
```

The CMC standard language does not support any physical units. Including such physical units can cause unintended values to be assigned to parameters. A warning message shall be sent when the parser detects

any characters after the unit-prefix, to warn the user that units are not understood by the parser and could cause unintended behavior.

The following examples show equivalent lines on either side:

`C1 (node1 0) capacitor C=1e-6farad`	`C1 (node1 0) capacitor C=1e-6`

`M1 (node1 0) mosfet L=0.01meter W=2umeter`	`M1 (node1 0) mosfet L=1e-5 W=2e-6`

4.1.7 Integer numbers

The CMC standard language does not allow the user to define the type of a number. Integer numbers are treated like real numbers with an integer value. Where the simulator expects an integer number, e.g. a model parameter, the simulator handles the conversion using the rules from the Verilog A LRM 2.3 [1]. The conversion happens by rounding the real number to the closest integer number. When the fraction is 0.5, round away from 0. Where the language needs to count an element of a vector, string or bus, integer numbers will be used.

//From A7.2.2

integer_number ::= [<sign>]<unsigned_number> | **int "("** <real_expression> **")"**

4.1.8 Boolean values

Boolean values are valued false or true. Numerically zero means false, while any other value means true.

//From A7.2.2

boolean_value ::= **true** | **false** | <real_number>

4.1.9 Names

A normal name may only contain letters, numbers and the characters "_", "#" and "!". The name can start with any of these characters, except “#”. But by enclosing the name in double quotes, any set of characters may be used. The quotes are not part of the name but just serve to delimit the node name. Names shall not be the same as reserved keywords. Implementations may set a limitation to the number of characters a name contains, but the limit shall be at least 1024 characters. Note that 0 is the reserved terminal for ground.

//From A7.3.1

name ::= <unquoted_name> | <quoted_name> excluding <keyword>
unquoted_name ::= <starting_name_character> { <name_character> }
quoted_name ::= **"""** <printable_ASCII_character> { <printable_ASCII_character> } **"""**
name_character ::= **a-z** | **A-Z** | **0-9** | **"_"** | **"#"** | **"!"**
starting_name_character ::= **a-z** | **A-Z** | **0-9** | **"_"** | **"!"**

In the following example, "node1" and node1 are different notations for the same terminal:

```
C1 (node1 0) capacitor C=1u
```

```
C2 ("node1" "here_#_is_a_$_strange node name") capacitor C=1u
```

4.1.10 Operators

The following operators are allowed on real numbers.

= assign value
+ addition
- subtraction
* multiplication
/ division
== comparison is equal to
< smaller than
<= smaller than or equal to
> larger than
>= larger than or equal to

//From A7.3.2

unary_operator ::= "-" | "+"
binary_operator ::= "+" | "-" | "*" | "/" | "=" "=" | "<" | "<" "=" | ">" | ">" "=" | "!" "="

The following operators are allowed on strings:
= assign value
== comparison, is equal
!= comparison, is not equal to

//From A7.3.2

string_operator ::= "=" "=" | "!" "="

The order of operation on the different operators will be the one described in Verilog A LRM 2.3 [1]. For reference, the order of operation is copied here from highest precedence to lowest.

+, - (unary)
*, /
+, - (binary)
<, <=, >, >=
==, !=
Ternary conditional operator ?:

Parentheses can be used to define the order of operation. Expressions within parentheses have to be evaluated first.

4.1.10.1 Ternary Conditional Operator

//From A7.3.2

real_ ternary_conditional_operation ::=

"(" <boolean_expression> **")"** **"?"** <real_expression> **":"** <real_expression> |
<boolean_expression> **"?"** <real_expression> **":"** <real_expression>
boolean_ternary_conditional_operation::=
"(" <boolean_expression> **")"** **"?"** <boolean_expression> **":"** <boolean_expression> |
<boolean_expression> **"?"** <boolean_expression> **":"** <boolean_expression>
string_ternary_conditional_operation::=
"(" <boolean_expression> **")"** **"?"** <string_expression> **":"** <string_expression> |
<boolean_expression> **"?"** <string_expression> **":"** <string_expression>

If the value of boolean_expression is true, then the return value of the function is the evaluated value of the first expression, otherwise it is the evaluated value of the second expression.

In the following example, if param is larger than 10, the variable Rvalue gets the value of param otherwise it gets the value of the double of param.

```
Rvalue = (param>10) ? param : 2*param
```

This operation is different than the conditional instantiation in 4.1.11 or 4.2.8, it only returns a value which depends on the evaluation of the boolean_expression. It can only be used where an expression can be placed. On the other hand if{}/else{} commands allows conditional instantiation of netlist lines. It does not return a value and it can only be used at a new line.

4.1.11 Functions

The CMC language supports the predefined functions listed later in this section as well as those defined by the user. A function is called by its name, which has to be a legal name and the function parameters come after the function call between parentheses "(" and ")" separated by a comma.

//From A7.3.3

function ::= <function_name> **"("** [<function_parameter_list>] **")"**
function_name ::= <name>
parameter ::= <real_number> | <boolean_value> | <string> | <vector> | <expression>
function_parameter_list ::= <parameter> {"," <parameter>}

4.1.11.1 User-defined functions

A user-defined function is defined by:

//From A7.3.3

function_definition ::= <function_name> **"("** [<parameter_name_list>] **")"** **"="** **\n**
"{" **\n**
[{<function_line> **\n**}]
return <expression> **\n**
"}" **\n**
function_line ::= <function_conditional> | <variable_assignment>
parameter_name_list ::= <parameter_name> { **","** <parameter_name>}

real_function_definition ::= <real_function_name> **"("** [<parameter_name_list>] **")" "="** **\n**
"{" **\n**
[{<function_line> **\n**}]
return <real_expression> **\n**
"}" **\n**
real_function_name ::= <name>
real_function_call ::= <real_function_name> **"("** [**<**function_parameter_list>] **")"**

boolean_function_definition ::= <boolean_function_name> **"("** [< function_parameter_name_list>] **")"**
"=" **\n**
"{" **\n**
[{<function_line> **\n**}]
return <boolean_expression> **\n**
"}" **\n**
boolean_function_name ::= <name>
boolean_function_call ::= <boolean_function_name> **"("** [**<**function_parameter_list>] **")"**

string_function_definition ::= <string_function_name> **"("** [<parameter_name_list>] **")" "="** **\n**
"{" **\n**
[{<function_line> **\n**}]
return <string_expression> **\n**
"}" **\n**
string_function_name ::= <name>
string_function_call ::= <string_function_name> **"("** [**<**function_parameter_list>] **")"**

function_name is the legal name of the user-defined function

parameter_name_list is a list of the names of parameters that are passed to the function by the user separated by a comma, these are the function arguments.

function_line is an optional set of expressions, assignments or conditionals.

return is a keyword to return the evaluated value of the expression which follows as the value of the function

When the **return** line is executed by the parser, the expression is evaluated and returned to the expression which called the function. The rest of the function is ignored.

```
var1 = 10k
var2 = if (var1 < 5) then 1 else 0 endif
var3 = var1+var2*log(10/3)
myMicro(x) =
{
return x*1e-6  // user defined function
}
R1 (node1 0) resistor R=var3+5u
C1 (node1 0) capacitor C=myMicro(3.3)
```

```
myMicro(x) =
{
var2 = if (x < 5) then 1 else 0 endif
```

```
var2 = exp(var2)
return x*var2*1e-6  // user defined function
}

R1 (node1 0) resistor R=5u
C1 (node1 0) capacitor C=myMicro(3.3)
```

```
sorta_invx(x) =
{
      if (x == 0)
      {
               return 0 //return 0 if x is 0
      }
      return 1/x //otherwise return 1/x
}
```

Functions have their own namespace.

It is possible to use local function-variables within a function, by assigning them a value. These variables are local to the function call only, they can be given a new value within a function, but do not affect any variable outside the function. Only local function-variables and the function arguments can be used within a function together with environment parameters (described in 4.2.9). Only the returned value of the function is returned to the instance which called for it. A function can contain other (user-defined) functions, but it can't call itself or another function which calls itself. Recursive functions are not allowed.

A function can contain a conditional statement as described in 4.2.8.

//From A7.3.3

function_conditional ::= **if "("** <boolean_expression> **")" \n**
"{" \n
{<function_line> **\n**}
"}" \n
[**else \n**
"{" \n
{<function_line> **\n** }
"}" \n]

---

if is a keyword, the boolean_expression is an expression which has a Boolean value, and which can be evaluated by the parser. Everything between the first set of curly braces "{" and "}" will be executed if the value of this expression is true. Otherwise, it will be ignored.

else is a keyword, everything between the curly braces "{" and "}" will be executed if the value of boolean_expression is false, otherwise it will be ignored.

4.1.11.2 Mathematical functions

The limitations of the domain and type of the output are referenced here.

The following functions are defined for real numbers and return real numbers, albeit some return real numbers with integer values. Trigonometric functions expect arguments in radians, not degrees, while inverse trigonometric functions return arguments in radians, not degrees.

//From A7.3.3.1

natural_logarithm_function ::= **ln "("** <x> **")"** returns the natural logarithm of x (x>0)
log10_function ::= **log10 "("** <x> **")"** returns the base-10 logarithm of x (x>0)
exponential_function ::= **exp "("** x **")"** returns the exponential function of x with no limiting for large values of x
limiting_exponential_function ::= **explim "("** <x> **","** <y> **")"** returns the exponential function of x with linear extrapolation for x>y. When x<=y, explim(x,y)=exp(x) else explim(x,y) = exp(y)*(1.0+x-y)
power_function ::= **pow "("** <x> **","** <y> **")"** returns the power of x to y (for x>0, all y, if x=0, y>0, if x<0, int(y))
sine_function ::= **sin "("** <x> **")"** returns the sine of x
cosine_function ::= **cos "("** <x> **")"** returns the cosine of x
tangent_function ::= **tan "("** <x> **")"** returns the tangent of x (x!= (2*n+1)*pi/2, with n an integer value)
inverse_sine_function ::= **asin "("** <x> **")"** returns the inverse sine of x (-1<=x<=1)
inverse_cosine_function ::= **acos "("** <x> **")"** returns the inverse cosine of x (-1<=x<=1)
inverse_tangent_function ::= **atan "("** <x> **")"** returns the inverse tangent of x
atan2_function ::= **atan2 "("** <x> **","** <y> **")"** returns the inverse tangent of y/x (all x and all y, with atan2(0,0) =0), this function takes the signs of x and y into account, such that atan2(-1,1)=3π/4 and atan2(1,-1)=-π/4.
hyperbolic_sine_function ::= **sinh "("** <x> **")"** returns the hyperbolic sine of x
hyperbolic_cosine_function ::= **cosh "("** <x> **")"** returns the hyperbolic cosine of x
hyperbolic_tangent_function ::= **tanh "("** <x> **")"** returns the hyperbolic tangent of x
inverse_hyperbolic_sine_function ::= **asinh "("** <x> **")"** returns the inverse hyperbolic sine of x
inverse_hyperbolic_cosine_function ::=**acosh "("** <x> **")"** returns the inverse hyperbolic cosine of x (x>=1)
inverse_hyperbolic_tangent_function ::= **atanh "("** <x> **")"** returns the inverse hyperbolic tangent of x (-1<x<1)
absolute_value_function ::= **abs "("** <x> **")"** returns the absolute value of x
square_root_function ::= **sqrt "("** <x> **")"** returns the square root of x (x>0)
db_function ::= **dB "("** <x> **")"** returns the value of x in decibel 20*log(x) (x>0)
integer_value_function ::= **int "("** <x> **")"** returns the integer value of x
floor_function ::= **floor "("** <x> **")"** returns the lower or equal integer value of x
ceiling_function ::= **ceiling "("** <x> **")"** higher or equal integer value of x
nearest_integer_function ::= **nint "("** <x> ")" returns the nearest integer value of x, if x is 0.5, the value is rounded down
sign_function ::= **sign"("** <x> **")"** returns sign of x, -1 or 1 if negative or positive value, 0 if x is 0
minimum_function ::= **min "("** <x> **","** <y> **")"** returns the minimum value of x and y
maximum_function ::= **max "("** <x> **","** <y> **")"** returns the maximum value of x and y
x::= <real_expression>
y::= <real_expression>

real_predefined_function ::= natural_logarithm_function | log10_function | exponential_function | limiting_exponential_function | power_function | sine_function | cosine_function | tangent_function | inverse_sine_function | inverse_cosine_function | inverse_tangent_function | atan2_function |

hyperbolic_sine_function | hyperbolic_cosine_function | hyperbolic_tangent_function |
inverse_hyperbolic_sine_function | inverse_hyperbolic_cosine_function |
inverse_hyperbolic_tangent_function | square_root_function | db_function |
integer_value_function | floor_function | ceiling_function | nearest_integer_function | sign_function |
minimum_function | maximum_function | absolute_value_function | range_check

The following functions are defined for strings:

//From A7.3.3.1

uppercase_function ::= **upper "("** <s> **")"** converts all lower case characters in string s to upper case
lowercase_function ::= **lower "("** <s> **")"** converts all upper case characters in string s to lower case
concatenate_function ::= **concat "("** <s> **","** <t> **")"** concatenates strings s and t with no intervening space
s::= <string_expression>
t::= <string_expression>

4.1.12 Expressions

Expressions can contain numbers, operators, functions and parameter names.

//From A7.3.4

expression ::= <real_expression> | <string_expression> | <boolean_value>
real_operation ::= [<unary_operator>] <real_expression> [<binary_operator> <real_expression>] | **"("** [<unary_operator>] <real_expression> [<binary_operator> <real_expression>] **")"**
string_operation ::= <string_expression> [<string_operator> <string_expression>] | **"("** <string_expression> [<string_operator> <string_expression>] **")"**
string_comparison ::= <string_expression> **"="** **"="** <string_expression> | <string_expression> **"!"** **"="** <string_expression>
real_expression ::= <real_number> | <real_operation> | <real_function> | <real_variable> | <real_parameter> | <string_comparison> | <real_ternary_conditional_operation>
boolean_expression ::= <real_expression> | <boolean_value> | <boolean_ternary_conditional_operation>
string_expression ::= <string> | <string_operation> | <string_function> | <string_variable> | <string_parameter> | <string_ternary_conditional_operation>
real_function ::= <real_predefined_function> | <real_function_call> | <statistical_function>
string_function ::= <string_predefined_function> | <string_function_call>
boolean_function ::= <boolean_predefined_function> | <boolean_function_call>

4.1.13 Vectors

Vectors or lists of values are defined between square brackets "[" and "]" and the values are separated by a comma. The values can be expressions of the same data-type.

//From A7.2.3

vector ::= real_vector | boolean_vector | string_vector
real_vector ::= **"["** <real_expression> { **","** <real_expression> } **"]"**
boolean_vector ::= **"["** **<**boolean_expression> | { **","** <boolean_expression> } **"]"**
string_vector ::= **"["** **<**string_expression { **","** <string_expression> } **"]"**

```
times = [0,1n,2n,5n,…]
```

4.1.14 Variables

Variables are symbols for values which can be used within expressions in the netlist. Parameters are symbols for values which are used to pass a value to an instance/model/subcircuit. Within a subcircuit, a parameter can be used in an expression (but its value cannot be redefined), while the value of a variable cannot be passed to the subcircuit from the outside.

Variables generally can be assigned a value once within a given scope. The variable assignment acts as the definition of the variable. Multiple assignments can be made on the same line in the netlist.

//From A7.3.5

variable_assignment ::= <single_variable_assignment> { <delimiter> <single_variable_assignment> }
single_variable_assignment ::= <real_variable_assignment> | <boolean_variable_assignment> | <string_variable_assignment> | <vector_variable_assignment>
real_variable_assignment := <real_variable> "=" <real_expression>
real_variable := <name>
boolean_variable_assignment := <boolean_variable> "=" <boolean_expression>
boolean_variable := <name>
string_variable_assignment := <string_variable> "=" <string_expression>
string_variable := <name>
vector_variable_assignment := <vector_variable> "=" <vector>
vector_variable := <name>

This will define a variable with the name real_variable, boolean_variable, string_variable or vector_variable with the respective value real_expression, boolean_expression, string_expression and vector throughout the scope the line is placed in. The variable can be used in any expression regardless of the location of the line of the assignment.

In the following examples, the netlist on the left hand side is equivalent to the one on the right hand side.

`variable1=x variable2=y` `variable3=z`	`variable1=x variable2=y` `+variable3=z`

The following lines generate a model of a resistor with r=x=1

`model Rmodel resistor r=variable1` `variable1=x`	`variable1=x` `model Rmodel resistor r=variable1`

A variable has one value in any given level of hierarchy. Within a given level of hierarchy, the user can assign a value to a variable only once, except by using a **redefine** keyword. If the user assigns a value to a variable more than once in a given scope without this keyword, the simulator shall send and error message to the *parser output* and the simulation ends.

This is illustrated in the following example which results in an error message being sent to the parser output that the variable x has been redefined.

```
x=1
r1 (node1 0) resistor r=x
x=2
r2 (node2 0) resistor r=x
```

The user has the capability of changing the value of a variable within a given scope by using the **redefine** command within that scope**.**

//From A7.3.5

redefine_variable ::= redefine_real_variable | redefine_boolean_variable | redefine_string_variable | redefine_vector_variable
redefine_real_variable ::= **redefine** <real_variable> "=" <real_expression>
redefine_boolean_variable ::= **redefine** <boolean_variable> "=" <boolean_expression>
redefine_string_variable ::= **redefine** <string_variable> "=" <string_expression>
redefine_vector_variable ::= **redefine** <vector_variable> "=" <vector_expression>

This command sets the value of the variable to that of the expression. This redefined value of the variable will be used throughout the scope regardless of the location in the netlist of the redefine line and the original assignment. This command can only be used once per variable in a given scope. If the user tries to redefine the same variable more than once in a given scope, the simulator shall send an error message to the *parser output* and the simulation ends. If the user tries to redefine a variable which has not been previously defined, the simulator shall send an error message to the *parser output* and the simulation ends.

Both of the following netlists instantiate resistors r1 and r2, both with r=2

```
x=1
r1 (node1 0) resistor r=x
redefine x=2
r2 (node2 0) resistor r=x
```

```
redefine x=2
r1 (node1 0) resistor r=x
x=1
r2 (node2 0) resistor r=x
```

The hierarchical and scoping rules for the value of a variable across different levels of hierarchy are discussed in 4.3.2 and 4.3.3.

4.1.15 Parameter assignments

Parameters can be explicitly passed down to an instance, model or subcircuit using parameter assignment on the instance line.

//From A7.3.6

parameter_list ::= <single_parameter_assignment> { <delimiter> <single_parameter_assignment> }
single_parameter_assignment ::= <real_parameter_assignment> | <boolean_parameter_assignment> | <string_parameter_assignment> | <vector_parameter_assignment>
real_parameter_assignment := <real_parameter> ["=" <real_expression>]
real_parameter := <name>
boolean_parameter_assignment := <boolean_parameter> ["=" <boolean_expression>]
boolean_parameter := <name>
string_parameter_assignment := <string_parameter> ["=" <string_expression>]

```
string_parameter := <name>
vector_parameter_assignment := <vector_parameter> [ "=" <vector> ]
vector_parameter := <name>
parameter = <real_parameter> | <boolean_parameter> | <string_parameter> | <vector_parameter>
```

Lists of parameter assignments can be created by separating multiple parameter assignments with a delimiter.

If an expression contains a parameter which at evaluation has not been defined in the netlist, then the parser returns an error to the *parser output*.

4.1.16 Terminals

Terminals are single nodes of the net, or collections of nodes (*bus terminal*). The list of *terminals* are optionally defined between parentheses "(" and ")", and separated by a delimiter.

Terminals are referred to by their name, which has to abide to the naming rules and cannot start with a "." and cannot contain any square brackets "[" and "]" . A list of terminals can be assigned to an instance in their standard order, or by their name. A model or a subcircuit can have a terminal which does not need to be passed to it, the language defines a mechanism to not pass a terminal to a model or subcircuit. When the reserved keyword **?UNCONNECTED** is passed to a terminal, that terminal is regarded as if it is not specified. To assign a terminal to a named node the following convention is used:

```
                                                                    //From A7.3.7
terminal_assignment ::= "." <name_of_terminal_in_model> "(" <terminal_name> ")"
name_of_terminal_in_model ::= <name>
terminal_name ::= <name> | "?"UNCONNECTED
```

name_of_terminal_in_model is the name of the terminal in the model/subcircuit/primitive that is being called

terminal_name is the legal name of the terminal in the current scope

If one terminal is assigned by name, all terminals have to be assigned by name. Terminals which are not explicitly assigned, are considered as not specified. An error shall be sent if the same terminal is assigned more than once.

If terminals are assigned as an ordered list, the reserved keyword **?UNCONNECTED** can be used to define that the terminal is not specified. If the model or subcircuit has more terminals than the ordered list, the parser will regard the missing terminals at the end to be unspecified.

A terminal list is defined as:

```
                                                                    //From A7.3.7
terminal_list ::= [ <terminal_name> ]{ <delimiter> <terminal_name> }  | "("
[ <terminal_name> ]{ <delimiter> <terminal_name> } ")" |  [ <terminal_assignment> ]{ <delimiter>
<terminal_assignment> }  | "(" [ <terminal_assignment> ]{ <delimiter> <terminal_assignment> } ")"
terminal ::= <terminal_assignment> | <terminal_name>
```

The number of terminals in the list shall not be larger than the number of terminals which have been declared in the underlying model/instance/subcircuit.

For example, a bipolar transistor with optional substrate and thermal terminals such as Mextram has 5 nodes named c b e s dt (defined by model developer).

```
q1a (.c(top) .b(top) .e(gnd) .dt(q1a_t)) mextram length=0.35u width=0.5u
```

The following three example boxes all feature instance lines which are equivalent to each other

```
nfet1 (d g s b) bsim4 w=1u l=0.1u
nfet1 (.drain(d) .gate(g) .source(s) .bulk(b)) bsim4 w=1u l=0.1u
nfet1 (.gate(g) .drain(d) .source(s) .bulk(b)) bsim4 w=1u l=0.1u
```

```
q1a (.c(top) .b(top) .e(gnd) .dt(q1a_t)) mextram504 area=1
q1a (top top gnd ?UNCONNECTED q1a_t) mextram504 area=1
```

```
q1b (.c(top) .b(top) .e(gnd) .s(sub)) mextram504 area=1
q1b (top top gnd sub ?UNCONNECTED) mextram504 area=1
q1b (top top gnd sub) mextram504 area=1
```

The following line shall result in an error as .drain is assigned twice

```
nfet1 (.drain(b) .gate(g) .drain(d) .source(s) .bulk(b)) bsim4 w=1u l=0.1u
```

4.1.16.1 Bus terminals

Bus terminals, which are terminal which contain a collection of nets instead of a single one. It allows instantiating multiple terminals to a single name. A terminal can be made into a bus terminal using square brackets "[" and "]" after the terminal name. The keyword **?UNCONNECTED** can also be used to explicitly not specify a bus terminal.

//From A7.3.7

bus_terminal ::= <terminal_name> "[" <start_integer> ":" <stop_integer> "]"
start_integer ::= <integer_number>
stop_integer ::= <integer_number>

---

instantiates a set of abs(stop_integer-start_integer)+1 terminals, which can be referenced after the definition by:

//From A7.3.7

bus_terminal_reference ::= <terminal_name> "[" <start_integer> [":" <stop_integer>] "]"

---

terminal_name is the name of a terminal which has been previously declared as a bus terminal.

If stop_integer is defined, a bus terminal with abs(stop_integer-start_integer)+1 terminals is referenced, where terminals start_integer through stop_integer are selected. If stop_integer is larger than

start_integer the terminals will be in ascending numerical order, otherwise they will be in descending numerical order.

If only start_integer is defined, the start_integerth terminal of the bus terminal is referenced, which will be treated as if it was not a bus terminal.

4.1.16.2 Assignment to bus instances

The terminal list for a *bus instance* can be assigned by list or by name.

By list, the following convention will be used, for a bus instance with n instances, the first n terminals will be assigned to the first terminal of every individual instance, the next n terminals to the next terminal and so on.

By name, if one terminal is assigned by name all terminals have to be assigned by name. A similar convention is used for a bus instance with n instances as with regular instances, except that the terminal_name is now a list of n terminals.

//From A7.3.7

busterminal_assignment ::= **"."** <name_of_terminal_in_model> **"("** <list_of_terminals> **")"**
list_of_terminals ::= <terminals> { <delimiter> <terminals> }
terminals ::= <terminal> | <bus_terminal>

---

name_of_terminal_in_model is the name of the terminal in the model/subcircuit/primitive that is being called

list_of_terminals is a list of n legal terminal names in the current scope

4.1.17 User-defined models and primitive models

User-defined models can be created using the model statement, as a Verilog-A module or as a C-API element. *Primitive models* are defined by a CMC standard or by Verilog-A code included in this document. The list of primitive models can be found in Appendix 2.

BSIM3 and BSIM4 have a different identifying name, bsim3 and bsim4. BSIM3 version 3.24 and version 3.23 do not. If the model developer would decide that they need a different name they can have that name added to the list of identifying names. The version and subversion number are defined by the model developers by using the CMC release procedures and shall be real numbers.

4.1.17.1 Verilog-A model

A file containing Verilog-A code can be loaded with the following compiler directive:

//From A7.4.1

load_veriloga ::= **import """veriloga""" "," """**<file_name>**"""**

---

import "veriloga" is a compiler directive to load a Verilog-A file

file_name is a string containing the name of the file, with the appropriate path, depending on the operating system, as described in 4.1.21.

Verilog-A models are called in the same way as a primitive or otherwise defined model.

A file containing compiled Verilog-A code can be loaded with the following compiler directive:

//From A7.4.1

load_compiled_veriloga ::= **import """compiled_veriloga""" "," """**<file_name>**"""**

---

import "compiled_veriloga" is a compiler directive to load a compiled Verilog-A file

file_name is a string containing the name of the file, with the appropriate path, depending on the operating system, as described in 4.1.21.

One file will not work for all simulation tools, as this is compiled code. This will not work for all operating systems. It is possible to resolve this by identifying the simulation environment as described in 4.1.18.1.1.

Verilog-A models shall be declared at the highest hierarchical level of a netlist. In general, Verilog- A models will be available globally throughout the netlist. A possible exception is to allow scoping of Verilog-A files when loaded within a technology. This allows the user to resolve potential naming conflicts when 2 design kits are used within the same netlist.

```
subckt subcircuit_1 (node1 node2)
import "veriloga" , "verilog_file.va" //this file contains module test
test1 (node1 node2) test //instantiates the test module
end subcircuit_1

test2 (node1 node2) subcircuit_1
test1 (node1 node2) test
```

4.1.17.2 C-API Models

C-API models are called in the same way as a primitive or otherwise defined model.

The TMI2 directory can be loaded using:

//From A7.4.1

load_tmi2 ::= **import """TMI2""" "," """**<path>**"""**

---

import "TMI2" is a compiler directive to point to a TMI2

path is a quoted string containing the appropriate path, depending on the operating system, to the TMI2 files.

The number of libraries which can be active at the same time within a given technology is set by the rules of the C-API.

4.1.18 C-preprocessor commands

C-preprocessor commands can be used in the CMC standard language. It is important to understand that these commands are statically defined. The CMC standard language supports the following C-

preprocessor commands: #define, #undef, #ifdef, #include, #if, #else, #elif #endif and –D on the command line. Their usage will follow the description in the C-standard ISO/IEC9899:TC2. The CMC language requires that the evaluation can occur by the parser and does not depend on any simulation variables such as voltages or currents. This section will explain the usage, the C-standard is the reference which shall be followed.

//From A7.4.2

preprocessor_define ::= **#define** <token1> [**"("** <parameter_name> { **","** <parameter_name>} **")"**]
[<token2>]
parameter_name ::= <token>
token1 ::= <identifier_token>
token2 ::= <token>
token::= <text_of_string>
identifier_token::= <identifier_token_character>{<identifier_token_character>}
identifier_token_character ::= <printable_ASCII_character> except <delimiter>

---

is a macro-definition. It replaces the token token1 by token2 throughout the netlist until it is terminated by a corresponding #undef. The pre-processor will add token1 to its namespace and its presence can be checked by #ifdef. If token2 is missing, all occurrences of token1 will be replaced by a single space. If parameters are present between the parentheses after token1, then the names of the parameters within token2 will be replaced by the arguments of the instance that is replaced by token1 in the netlist.

```
#define CHECK 0
#define sum(a,b) a+b

test=sum(1,CHECK)
```

will first be replaced by

```
test=sum(1,0)
```

and then by

```
test=1+0
```

//From A7.4.2

preprocessor_undefine ::= **#undef** <token1>

Terminates a previously defined macro-definition, which has been defined by #define token1 […]. It removes the macro from the namespace and the previously defined macro will not replace anything else beyond this in the netlist. If token1 has not been previously defined this is ignored.

In the following example, the value of test is 0 and the value of test2 is the string CHECK.

```
#define CHECK 0
test=CHECK
#undef CHECK
test2=CHECK
```

//From A7.4.2

preprocessor_ifdef::= **#ifdef** <token1> **\n**
{ <statement> }
[**#else** { <statement> } **\n**]
#endif \n
preprocessor_ifndef::= **#ifndef** <token1> **\n**
{ <statement> }
[**#else** { <statement> } **\n**]
#endif \n

Tests if the macro token1 is present for #**ifdef**, or not present for #**ifndef** in the macro namespace. If this is the case, then the lines between the #**ifdef** or #**ifndef** statement to the following **#else** or #**endif** statement are processed by the parser, otherwise they are ignored. If the block was ended by **#else** then the block between **#else** and **#endif** will be executed if the previous block was not, otherwise all lines between **#else** and **#endif** will be ignored.

In the following example, the value of check is 0 at the end.

```
#define CHECK 0
#ifdef CHECK
check=1 //is parsed
#endif

#undef CHECK
#ifdef CHECK // is false
check=1.5 //is ignored
#endif

#ifndef CHECK // is false
check=0 //is parsed
#endif
```

//From A7.4.2

preprocessor_include ::= **#include** """ <file_name> """

The CMC standard language allows the use of this command to include a file, just like in the C-standard. This is described in 4.1.22. An extension of this command to include libraries can be found in 4.2.4. The file_name is a quoted string.

//From A7.4.2

preprocessor_if ::= **#if** <boolean_expression> **\n**
{ <statement> }
{**#elif** <boolean_expression> **\n**
{ <statement> }
[**#else \n**
{ <statement> }]
#endif \n

The parser must be able to evaluate the boolean_expression during parsing. They cannot depend on any information from the simulator.
If the first boolean_expression evaluates to a non-zero value (true), then all lines until the following **#elif**, **#else** or **#endif** are executed, and if this is not an **#endif** statement all lines from then to the **#endif** statement are skipped.
If the first boolean_expression evaluates to a zero value(false), if the next statement is **#elif**, if the second boolean_expression is evaluated as true, then the lines until the following **#elif**, **#else** or **#endif** are executed and if this is not an **#endif** statement all lines from then to the **#endif** statement are skipped. This is repeated with multiple **#endif** statements.
If all previous **#if** and **#elif** statements are evaluated as false, then all the lines between **#else** statement and **#endif** are executed.

The following example will replace all further occurrences of TEST by 1:

```
#define CHECK 1

#if CHECK>1
#define TEST 2
#elif CHECK>0
#define TEST 1
#else
#define TEST 0
#endif
```

-D on the command line
The CMC standard language allows the user to define a macro on the command line when invoking the simulation program or the parser. Placing –D <token1> [= <token2>], on the command line will place the corresponding: **#define** <token1> [<token2>] at the top of the called netlist. Token2 can only be a constant integer or character. If the macro has already been defined within the netlist this will result in a warning, and the original definition (already in the netlist) will be used.

```
simulator.exe -DMAC=50 circuit.ckt
```

is the same as adding the following line to the top of the netlist circuit.ckt:

```
#define MAC 50
```

4.1.18.1 Reserved Macros

Appendix 6 contains a list of the reserved macros. These cannot be defined by the user, but are defined by the simulator. Reserved macros start with "_".

4.1.18.1.1 Simulation environment

There might be a need to tailor a netlist to the simulator which uses it. The C-preprocessor commands can be used to instantiate parts of the netlist depending on the simulator, by using reserved macros defined by every simulator vendor and the reserved macros _MAJOR_VERSION and _MINOR_VERSION. The name of the simulator-specific identifying macro shall be defined and documented by the creator of the tool, it shall follow the rules of a reserved macro and start with a "_".

In the following example, if the _EDA_TOOL macro has been defined to be "EDA_simulator", three different lines can be executed depending on the definition of the _MAJOR_VERSION macro. If it is 1the options setting=1.0 line is executed, if it is 2, the options setting=2.0 line will be executed, else the options setting=0.0 line is executed.

```
#ifdef _EDA_TOOL
#if _MAJOR_VERSION == 1
options setting=1.0
#elif _MAJOR_VERSION == 2
options setting=2.0
#else
options setting=0.0
#endif
```

There might also be a need to change the netlist to the operating system which is being used. If the simulator runs in Windows then **#ifdef** _WINDOWS will be true. If it runs in Linux, **#ifdef** _LINUX will be true and for Solaris **#ifdef** _SOLARIS will be true

In the following example, if the simulator runs in a Windows environment the file "C:\WindowsPath\Includefile.txt will be included, while if it would run in a Linux environment, the file at "~/linuxpath/Includefile.txt" would be included.

```
#ifdef _WINDOWS
#include "C:\WindowsPath\Includefile.txt
#endif
#ifdef _LINUX
#include "~/linuxpath/Includefile.txt"
#endif
```

4.1.19 Interface with other languages

The CMC standard language allows part of the netlist to be written in another language. The keyword **simulatorlanguage** can be used to switch between languages.

//From A7.4.3

set_simulatorlanguage ::= **simulatorlanguage "="** <language_name>
language_name ::= **cmc_standard** | **VerilogA** | <string>

---

simulatorlanguage is a keyword which allows all following lines to be read as if they were written in the language language_name. The parser will assume this remains so until another simulatorlanguage command is used.

The CMC standard language will be invoked as **simulatorlanguage** = cmc_standard.
Verilog A will be invoked as **simulatorlanguage** = VerilogA.

When the language is set by the simulatorlanguage command, the rules of that language will be used until the next simulatorlanguage command is found by the parser. This is persistent through include files.

A functionality of the interface may be defined between the cmc_standard language and the other languages by the owner of the other language.

4.1.20 User-generated warnings and errors

The user has the ability to have a warning printed to a *parser output*. This output can be a message to the screen or to a log file, this depends on the simulator environment. This warning can contain the value of any expression.

//From A7.4.4

warning ::= **warning "("** <warning_message> [{"," <argument>}] **")"**
warning_message ::= <string>
argument ::= <expression>

---

warning is a keyword which allows the user to print a text to the parser log.

warning_message is a string, %e, %E, %f, %F, %g, %G, %m, %M, %s and %S can be used to insert one of the arguments into the string. The first %e will be replaced by the first argument and so on. %e, %E, %f, %F and %g, %G are used with real numbers and support the formatting options of C. By placing 2 integers separated by a "." ahead of the letter, the minimum field width and the number of decimal fractional digits can be set. %m or %M allows the user to print out the hierarchical name of the instance it refers to. %s or %S is used with a string.

arguments are expressions separated by a comma

The CMC standard language also has a function to do range checking for parameters in a subcircuit call.

//From A7.4.4

range_check ::= **range_check "("** <input> **","** <lower_bound> **","** <upper_bound> **","**
<warning_message> **")"**
input ::= <real_expression>
lower_bound ::= <real_expression>
upper_bound ::= <real_expression>

---

range_check is a keyword for the function which returns the value of a parameter bounded by a minimum and a maximum value

If both lower_bound and upper_bound are assigned a value, upper_bound shall be greater than lower_bound, otherwise an error message will be sent by the parser.

If a value of lower_bound is passed to the function and if input is smaller than lower_bound, then the function returns is the lower_bound and the warning_message is printed to the parser output. If a value of upper_bound is passed to the function and if input is greater than upper_bound, then the function returns the maximum_value and the warning_message is printed to the parser output. Otherwise the function returns the input. If the input cannot be evaluated, an error message is sent to the parser output, similar to any other expression which cannot be evaluated.

In the following example, x=0.6, y=5, z=0.6, z2=1, z3=0, z4=-0.4 and z5=5, the warning messages "The value of y = 5.000000e+00 must be between 0 and 1" and "The value of x-1 = -4.000000e-01 must be between 0 and 1" will be sent to the parser output.

```
x=0.6
y=5
z=range_check(x,0,1,"The value of x = %e must be between 0 and 1",x)
z2=range_check(y,0,1,"The value of y = %e must be between 0 and 1",y)
z3=range_check(x-1,0,1,"The value of x-1 = %e must be between 0 and 1",x-1)
z4=range_check(x-1,,1,"The value of x-1 = %e must be smaller than 1",x-1)
z5=range_check(y,0,,"The value of y = %e must be greater than 0 and 1",y)
```

The user may have the option to set the number of times an identical warning shall be printed to the *parser output*.

The user also has the ability to instantiate an error, with a message to the *parser output,* which also causes the parser to exit after parsing the rest of the netlist, but before starting the simulation. This output can be a message to the screen or to a log file, this depends on the simulator environment. This warning can contain the value of any expression.

//From A7.4.4

error ::= **error "("** <warning_message> [{"," <argument>}] **")"**

---

error is a keyword which allows the user to print a text to the parser log.

warning_message is a string, %e, %E, %f, %F, %g, %G, %m, %M, %s and %S can be used to insert one of the arguments into the string. The first %e will be replaced by the first argument and so on. %e, %E, %f, %F and %g, %G are used with real numbers and support the formatting options of C. By placing 2 integers separated by a "." ahead of the letter, the minimum field width and the number of decimal fractional digits can be set. %m or %M allows the user to print out the hierarchical name of the instance it refers to. %s or %S is used with a string.

arguments are expressions separated by a comma

In the following example, the warning messages " x is too large" and "y is too large" will be sent to the parser output and the simulation will terminate after parsing.

```
x=2
y=3
if(x>1)
{
error("x is too large")
}
if(y>1)
{
error("y is too large")
}
```

The user also has the ability to instantiate an error, with a message to the *parser output,* which also causes the parser to exit immediately upon detection. This output can be a message to the screen or to a log file, this depends on the simulator environment. This warning can contain the value of any expression.

//From A7.4.4

fatal ::= **fatal "("** <warning_message> [{"," <argument>}] **")"**

---

fatal is a keyword which allows the user to print a text to the parser log.

warning_message is a string, %e, %E, %f, %F, %g, %G, %m, %M, %s and %S can be used to insert one of the arguments into the string. The first %e will be replaced by the first argument and so on. %e, %E, %f, %F and %g, %G are used with real numbers and support the formatting options of C. By placing 2 integers separated by a "." ahead of the letter, the minimum field width and the number of decimal fractional digits can be set. %m or %M allows the user to print out the hierarchical name of the instance it refers to. %s or %S is used with a string.

arguments are expressions separated by a comma

In the following example, the warning messages " x is too large" will be sent to the parser output and the simulation will terminate after it has read the fatal("x is too large") line.

```
x=2
y=3
if(x>1)
{
fatal("x is too large")
}
if(y>1)
{
error("y is too large")
}
```

4.1.21 Files

The language allows files to be referred to by the file_name. The file_name is a string containing the name of the file with the appropriate path. Either a relative path or an absolute path can be used. The syntax of the path name follows the convention of the operating system which the simulation tool uses. A reserved macro can be used to define a path differently depending on the operating system which is used. _WINDOWS, _LINUX and _SOLARIS, will be considered defined by the parser if the simulator is running in respectively Windows, Linux and Solaris. If a relative path is used, the working directory is the directory which contains the netlist, where the line which refers to the file_name is found. The working directory changes when a new file is parsed which is in a different directory, when the new file is finished parsing, the working directory moves back to the original one as the rest of the original file is being parsed. Keep in mind that the \ character is written as \\ in the string.

//From A7.4.1

path ::= <string>

4.1.22 Including netlist

The CMC standard language allows incorporating a different netlist into the netlist using the #**include** pre-processor command. The parser replaces the instance line of the include statement with the new netlist. This describes the same functionality as described in 4.1.18 above.

//From A7.4.2

preprocessor_include ::= **#include """** <file_name> **"""**

#include is a keyword to load a netlist

file_name is a quoted string containing the name of the file, with the appropriate path, depending on the operating system, as described in 4.1.21.

The language only supports including one netlist at a time. Other files in the directory of the included file are not loaded by the parser or searched for potentially missing model cards.

4.1.23 Namespace

The names of different aspects of a netlist have a different namespace. The following table shows the different namespaces that are allowed in the language and which aspects share them.

Names of variables, subcircuit parameters, paramsets, environment parameters and functions
Names of instances and bus instances
Names of models (also Verilog-A models), subcircuits
Names of terminals and global nodes
Names of sections of libraries
Filenames
Names of Technologies
Names of preprocessor-defined macros

If an identical name has already been defined within the same namespace, subsequent definition of this name will be flagged as an error by the parser. If a name is defined which only differs with an existing name (within the same namespace) in capitalization, a warning shall be issued to the *parser output*. Objects within different namespaces can share the same name within the same hierarchy.

The following example results in an error:

```
x1=1
paramset x1 paramset
{
y=1
}
```

While the next example results in a warning: "R1 and r1 only differ in capitalization within the same namespace":

```
model r1 resistor r=10
subckt R1 (node1 node2)
r1 (node1 node2) resistor r=10
end r1
```

The first three of these namespaces, can have a local copy within a particular scope. The others can only be globally defined. This is described in 4.3.3. The hierarchically lower subcircuits will inherit the higher

defined namespace, but new variables, models and instances can be re-defined with previously existing names.

4.1.24 Encryption directives

Some tools allow users to encrypt parts of the netlist. This requires two keywords which start and end the part which needs to be decrypted.

//From A7.4.5

encryption ::= **encrypted \n**
{ <line> }
endencrypted \n
line::= { <any_character> }
any_character ::= ASCII : 0x00-0xFF

encrypted and **endencrypted** are two reserved keywords which tell the encryption tool to encrypt everything between encrypt and endencrypt. This standard does not define any sort of encryption method; the encryption method is implementation dependent. The standard does not define the format of the encrypted content.

The parser can use these keywords to decrypt the part of the netlist between them.

4.2 Circuit Description

4.2.1 Instance statements

An *instance* is a particular placement of a *device* or *subcircuit*, represented in the netlist by an *instance statement***:**

//From A7.5.1

instance ::= <instance_name> <terminal_list> <instance_type> [<parameter_list>]
instance_name ::= <name>
instance_type := <model_name> | <subcircuit_name> | <string_parameter>

The instance_name has to be a legal name for an instance
terminal_list is a list of terminals, as defined in 4.1.16.
instance_type is the legal name of a primitive model, a user-defined model, a subcircuit or a string_parameter. A list of primitive models is found in Appendix 2.
parameter_list is a list of parameter assignments, as defined in 4.1.15

The following statement instantiates a resistor named "R1" between terminals node1 and node2 with a value of the parameter r of 50.

```
R1 (node1 node2) resistor r=50
```

It will not be possible to identify a device by merely having its name start with a certain letter, such as a capacitor by having its name start with the letter C. It is necessary to identify it as a capacitor on the instance line, if no user-defined model is used.

4.2.1.1 Bus instances

Bus instances are instances which contain multiple instances, but are referenced by a single instance name and are instantiated on a single instance statement. An instance can be made into a bus instance using square brackets "[" and "]" after the instance name.

//From A7.5.1

bus_instance ::= <instance_name> **"["** <start_integer> **":"** <stop_integer> **"]"** <terminal_list> <instance_type> [<parameter_list>]

---

instance_name is a legal instance name
start_integer and stop_integer are integer values
terminal_list is a list of terminals, as defined in 4.1.16.
instance_type is the legal name of a primitive model, a user-defined model, a subcircuit or a string_parameter. A list of primitive models is found in Appendix 2.
parameter_list is a list of parameter assignments, as defined in 4.1.15

will instantiate a set of abs(stop_integer-stop_integer)+1 instances, which can be referenced after the definition by:

//From A7.5.1

bus_instance_reference ::= <instance_name> "[" <start_integer> [":" <stop_integer>] "]"

---

instance_name is the name of a terminal which has been previously declared as a bus terminal.
If stop_integer is defined, a bus instance with abs(stop_integer-start_integer)+1 instances is referenced, where instances start_integer through stop_integer are selected. If stop_integer is larger than start_integer the instances will be in ascending numerical order, otherwise they will be in descending numerical order.
If only start_integer is defined, the start_integerth instance of the bus instance is referenced, which will be treated as if it was not a bus instance.

The following statements instantiates five resistors named "R1[i]" between terminals node1[i] and node2[i] with a value of the parameter r of 50:

```
R1[1:5] (node1[1:5] node2[1:5]) resistor r=50
```

4.2.2 Model statements

A *model* defines the equations defining terminal characteristics of an instance of a *device*, in terms of solution variables, *instance parameter* values, *model parameter* values, and *environment parameter* values. A *model statement* is used to give *model parameter* values for the *model* which describes a device. A model statement defines a name for itself (the *model name*):

//From A7.5.2

model ::= **model** <model_name> <device_type> [<parameter_list>]
model_name ::= <name>
device_type ::= <model_name> | <model_primitive> | <string_parameter>

---

model_primitive ::= psp | hisim | bsim3 | bsim4 | bsimsoi | hisim_hv | hicuml0 | hicuml2 | mextram504 | vbic | sgp | resistor | capacitor | inductor | mutual_inductor | diode_cmc | juncap2 | r2_cmc | r3_cmc | mosvar_cmc | vsource | isource | vcvs | vccs | ccvs | cccs | <string>

model is a keyword.

model_name is the legal name of this user-defined model.

device_type is a primitive, another user-defined model or a string_parameter. A list of primitive models is found in Appendix 2. If a model calls another model, it will overwrite the values of the parameters of the model it calls.

parameter_list is a list of parameter assignments, as defined in 4.1.15

The following statement creates a model named "Rmodel" which is a resistor with a default value of 50 for the parameter r:

```
model Rmodel resistor r=50
```

In the following example, bsim4_model_2 and bsim4_model_1 will have all parameters in common except vto, which will be 1 for bsim4_model_2 and -0.5 for bsim4_model_1.

```
model bsim4_model_2 bsim4_model_1 vto=1
model bsim4_model_1 bsim4 vto=-0.5 type=n
```

4.2.2.1 Automatic Model Selection (Binning)

4.2.2.1.1 Lmin/Lmax/Wmin/Wmax

The language allows one model to contain multiple sets of parameters. Each of these sets is linked to a particular bin, a certain range of width and length for the device. When a device is instantiated the first set of parameters, for which the length and the width are within the bin, is chosen by the parser.

//From A7.5.2

binned_model ::= **model** <model_name> <device_type> **\n**
"{" \n
<binning_label> **":"** [<parameter_list>] **\n**
{ <binning_label> **":"** [<parameter_list>] **\n** }
"}" \n
binning_label ::= <string>

model is a keyword.

model_name is the legal name of this user-defined model.

device_type is a primitive or another user-defined model. A list of primitive models is found in Appendix 2. If a model calls another model, it will overwrite the values of the parameters of the model it calls.

parameter_list is a list of parameter assignments, as defined in 4.1.15

binning_label is a name for the bin. This name can contain numbers and letters only. The binning_label needs to be unique for all bins within a certain model_name.

When an instance with parameter l and w calls this model the first bin, in the order in which they are instantiated, which meets:

lmin <= l < lmax and wmin <= w/nf < wmax or lmin == l == lmax and wmin == w/nf == wmax

Where lmin, lmax, wmin and wmax are model parameters that define the ranges for each bin and l and w are the instance length and width and nf is the number of fingers. The bin-specific parameters will overwrite model parameters if the device_type is another model. When the device dimensions do not fit in any bin, an error message will be sent by the parser.

```
// Define models with the global parameters.
model  nch bsim4
{
big: wmin=1e-6 wmax=1 lmin=0.2e-6 lmax=1 vth0=n1 u0=n11 ...
thin: wmin=0.1e-6 wmax=1e-6 lmin=0.2e-6 lmax=1 vth0=n2 u0=n21...
short: wmin=1e-6 wmax=1 lmin=0.05e-6 lmax=0.2e-6 vth0=n3  u0=n31...
...
}

M1 (n1 n2 n3 n4) nch l=0.5u w=0.5u //will have parameters from bin thin
M2 (n1 n2 n3 n4) nch l=0.08u w=2u //will have parameters from bin short
```

```
// Define models with the global parameters.
model  nch_global   bsim4  type=n   toxe=3e-9  k1=0.5  k2=-0.05 ...
model  nch   nch_global
{   // All bins are derived from nch_global. No need to respecify
// type, toxe, k1, k2,... (unless you want to override them).
1: wmin=1e-6 wmax=1 lmin=0.2e-6 lmax=1 vth0=n1 u0=n11...
2: wmin=0.1e-6 wmax=1e-6 lmin=0.2e-6 lmax=1 vth0=n2 u0=n21...
3: wmin=1e-6 wmax=1 lmin=0.05e-6 lmax=0.2e-6 vth0=n3 u0=n31...
...
}

M1 (n1 n2 n3 n4) nch l=0.5u w=0.5u //vth0=n2 u0=n21 toxe=3e-9 k1=0.5 k2=-0.05
M2 (n1 n2 n3 n4) nch l=0.08u w=2u // vth0=n3 u0=n31 toxe=3e-9 k1=0.5 k2=-0.05
```

4.2.2.1.2 Binning using any parameter

The CMC standard Spice language supports an alternative method of binning a model. This method is exclusive from the one described in 4.2.2.1.1, if this method is used lmin, lmax, wmin and wmax are not used for binning.

//From A7.5.2

binned_model_2 ::= **model** <model_name> <device_type> **\n**
"{" \n
<binning_label> **":"** <binning_condition> {<binning_condition>} [<parameter_list>] **\n**
{ <binning_label> **":"**<binning_condition> {<binning_condition>} [<parameter_list>] **\n** }
"}" \n
binning_condition ::= **<**parameter_name> **from** <enclosure_start> <real_expression> **":"**
<real_expression> <enclosure_end>
enclosure_start ::= **"["** | **"("**
enclosure_end ::= **"]"** | **")"**

The parser selects the first bin for which all instances of the binning_condition are evaluated as being true, in the order in which they are instantiated. The parameter_name is a parameter on the instance line which calls the model. The square brackets for enclosure mean that the binning_condition is met if the value of parameter_name is equal to the real_expression. When no bin is found where all the binning conditions are true, an error message will be sent by the parser.

With this method, the name can be overloaded to allow a different device_type to be used. The order of evaluation occurs in the order in which the overloaded definitions are read by the parser. The binning_label shall be unique for all bins with a certain model_name.

```
// Define models with the global parameters.
model  nch bsim4
{
thin: w from [0.1e-6:1e-6) l from [0.2e-6:1) vth0=n2 u0=n21...
short: w from [1e-6:1) l from [0.05e-6:0.2e-6) vth0=n3  u0=n31...
...
}

model  nch bsim3
{
big: w from [1e-6:1) l from [0.2e-6:1) vth0=n1 u0=n11 ...
}

M1 (n1 n2 n3 n4) nch l=0.5u w=0.5u //will have parameters from bin thin
M2 (n1 n2 n3 n4) nch l=0.08u w=2u //will have parameters from bin short
M3 (n1 n2 n3 n4) nch l=2u w=2u //will have parameters from bin big generating a device of the type bsim3
```

4.2.3 Subcircuit definition

A *subcircuit* is a representation of a group of devices, done to allow replication of the group of devices. The *subcircuit* is defined by a *subcircuit definition*, and is placed in the *netlist* using an *instance statement*. The *subcircuit definition* consists of statements beginning and ending the subcircuit definition, which declares the name for itself (the *subcircuit name*):

//From A7.5.3

subcircuit ::= **subckt** <subckt_name> < subcircuit_terminal_list> **\n**
{**parameters** <parameter_list> **\n** }
{ <statement> }
end <subckt_name> **\n**
subckt_name ::= <name>
subcircuit_terminal_list ::= [<subcircuit_terminal_name>]
{ <delimiter> <subcircuit_terminal_name> }
  | "(" [<subcircuit_terminal_name>]{ <delimiter>
<subcircuit_terminal_name> } ")"
optional_terminal ::= <terminal_name> "(" <terminal_default> ")"
subcircuit_terminal_name ::= <terminal_name> | <optional_terminal>
terminal_default ::= <previously_defined_terminal_name> | **"0"** | <global> | **"?"UNCONNECTED**
previously_defined_terminal_name ::= <terminal_name>

---

subckt is a keyword to define the start of the sub-circuit.
subckt_name is a legal name of the subcircuit

subcircuit_terminal_list is a list of terminals, which can also contain optional terminals. When a terminal is not passed to an optional terminal when the subcircuit is instantiated, the terminal defaults to the terminal_default.

terminal_default is the name of the terminal which the terminal defaults to, when it is unconnected. This name has to be another terminal for the subcircuit, which has been defined earlier on the line, a global node, the ground 0, or unconnected with the keyword **?UNCONNECTED**.

previously_defined_terminal_name is a terminal name which has been defined in the subcircuit definition in front of the optional_terminal for which it is set as a default.

parameters is a keyword which allows the user to define a set of parameters which can be passed down to the subcircuit. These parameters do not need to be assigned a default value. Multiple statements starting with **parameters** are allowed. The parameter list needs to form a contiguous block right after the **subckt** line. Every parameter may only be defined once.

The subcircuit can contain instances, models, variables, function definitions, other subcircuit definitions.

end is a keyword which sets the end of the subcircuit definition

A terminal which does not have a default value is required. If no terminal is specified for such a terminal, an error will be sent to the output. When an optional terminal defaults to being unconnected, the terminal will be treated as if it is internal to the subcircuit.

The following example defines a subcircuit consisting of a parallel resistor R1 and capacitor C1. The value of the resistance of R1 has to be set by the user, while the capacitance of C1 has a default of 1u.

```
subckt rc (node1 node2)
parameters r c=1u
R1 (node1 node2) resistor r=r
C1 (node1 node2) capacitor c=c
end rc
```

```
subckt mysubcircuit (node1 node2)
  // arg1 has a default value
  // arg2 has no default so it must be specified
  parameters arg1=3 arg2
  parameters arg3=1
  // size is a local variable to the subcircuit
  // and not a parameter that can be passed in
  size=arg2*1.5*arg3
  cmp1 (node1 node2) resistor r=arg1
  // ordinarily names can't start with a number
  // but we can overcome those restrictions by
  // quoting any arbitrary string
  model "1n914" diode is=1e-14 cjo=1p rs=10
  ClipDiode1 (node2 0) "1n914" area=size
end mysubcircuit

inst1 (node node2) mysubcircuit arg2=30 // use default arg1
```

The parameter list of the subcircuit needs to be a contiguous block right behind the line which contains the definition of the subcircuit. The simulator shall send an error message if anything but white space separates the parameter definition from the **subckt** line or other parameter definitions.

The following two examples result in an error message being sent:

```
subckt mysubcircuit (node1 node2)
  // COMMENT
  size=arg2*1.5
  parameters arg1=3 arg2 //this parameter definition can't be separated from the
subckt line by a variable assignment
  model "1n914" diode is=1e-14 cjo=1p rs=10
  ClipDiode1 (node2 0) "1n914" area=size
end mysubcircuit
```

```
subckt mysubcircuit (node1 node2)
  parameters arg2
  model "1n914" diode is=1e-14 cjo=1p rs=10
  parameters arg1=3 //this parameter definition can't be separated from the other
parameter definitions line by a model definition

  size=arg2*1.5
  ClipDiode1 (node2 0) "1n914" area=size
end mysubcircuit
```

A parameter can only be defined once in a given subcircuit. The simulator shall send an error message when two parameters with the same name are defined in the same subcircuit. Within a subcircuit variables and parameters share the same namespace. The simulator shall send an error message when a variable is defined (or redefined) in a subcircuit with the same name as one of the subcircuit parameters. There is no difference between parameters and variables at any hierarchy below the subcircuit.

The following two examples result in an error message being sent:

```
subckt mysubcircuit (node1 node2)
  parameters arg1=3 arg2 arg1=2 //arg1 cannot be defined twice
  model "1n914" diode is=1e-14 cjo=1p rs=10
  ClipDiode1 (node2 0) "1n914" area=1f
end mysubcircuit
```

```
subckt mysubcircuit (node1 node2)
  parameters arg2 arg1=3
  size=arg2*1.5
  arg2 = 5 //arg2 is a parameter, so it can't be defined as a variable
  model "1n914" diode is=1e-14 cjo=1p rs=10
  ClipDiode1 (node2 0) "1n914" area=size
end mysubcircuit
```

A parameter does not need to have a default value. If an instance calls a subcircuit without assigning a value to a parameter without a default value an error message shall be sent and the simulation will end.

The following example will result in an error message being sent:

```
subckt mysubcircuit (node1 node2)
  parameters arg1=3 arg2
  model "1n914" diode is=1e-14*arg2 cjo=1p*arg1 rs=10
  ClipDiode1 (node2 0) "1n914" area=1f
end mysubcircuit
```

```
subcircuit1 (1 0) mysubcircuit arg2=1 //is fine
subcircuit2 (2 0) mysubcircuit arg1=2 //generates an error as arg2 is undefined
```

The following example illustrates some of the possible options one has to instantiate a subcircuit:

```
subckt mysubcircuit (node1 node2(node1) node3(?UNCONNECTED) node4(0))
  mosfet (node1 node2 node3 node4) bsim4
end mysubcircuit
```

Subcircuit instantiation	Resulting device instantiation
`sub1 (1 2 3 4) mysubcircuit`	`sub1.mosfet (1 2 3 4) bsim4`
`sub1 (1 ?UNCONNECTED ?UNCONNECTED 4)`	`sub1.mosfet (1 1 sub1.node3 4) bsim4`
`sub1 (1 2) mysubcircuit`	`sub1.mosfet (1 2 sub1.node3 0) bsim4`
`sub1 (.node1(1) .node3(1)) mysubcircuit`	`sub1.mosfet (1 1 1 0) bsim4`
`sub1 (.node2(1) .node3(1)) mysubcircuit`	Results in an error as node1 is required

4.2.4 Library definition

A *library* is a collection of subcircuits, models, variables and user-defined functions for which the content changes depending on the *section*. The names of the subcircuits, models, variables and user-defined functions might be the same across sections. A library file will be a file which contains sections of the library.

The user can define a section in a library file <library_filename> with the keywords **section** and **endsection**:

//From A7.5.4

section ::= **section** <section_name> **\n**
{ <statement> }
endsection <section_name> **\n**
section_name ::= <name>

---

section and **endsection** are keywords, which define the start and ending of a block which will be executed if section_name is defined in the include_library.

section_name is the name of the section.

The section of the library can then be included in the netlist which uses the library, using the following commands:

//From A7.5.4

include_library ::= **#include """** <library_filename> **"""** **section "="** <section_name> **\n**
library_filename ::= <file_name>

#include is a keyword to include a library file

Library_filename, a quoted string, is the FILENAME of the library file, including the path, this is dependent of the operating system, described in 4.1.21.

section is a keyword to include a particular section of the library

section_name is the name of the section, which has to be included, its name has to follow the rules which govern names

If the section name cannot be found in the file, an error message shall be sent with a warning to the *parser output*.

4.2.5 Technologies

A *technology* is created by bookending the file or piece of it with the keywords **technology** and **endtechnology**. Only one technology can be active at any given point. If a new technology, is loaded, all data from a previous technology is cleared. A technology can contain most language structures, even those usually reserved for the top-level of the design. A technology can't contain another technology though.

A technology allows the user to create a local environment which is separate from the rest of the netlist. This acts as a separate top-level hierarchy. The user can redefine global quantities such as simulator options or global variables local to the technology. The technology inherits the global options from the highest hierarchy of the design, but has the capability of redefining them locally, without changing them at the higher level.

A technology will share the following namespace with other technology files and any hierarchy outside of technologies.

Instances and bus instances
Terminals and global nodes
Preprocessor-defined macros

No new hierarchy is created when it comes to the instance names or the terminals. The topological structure stays the same in and out of technologies; it is only the parameterization which has its own hierarchy.

A technology will generate its own local copy of the following namespace:

Variables, subcircuit parameters, paramsets and environment parameters
Models (also Verilog-A models), subcircuits
Functions

The technology is defined by:

//From A7.5.5

technology ::= **technology** <name_of_technology> **\n**

[<content_of_technology>]
endtechnology <name_of_technology> **n**
name_of_technology ::= <name>
content_of_technology ::= {<statement>}

technology and **endtechnology** are keywords used to signify the start and end of the technology file name_of_technologyfile. The technology extends from the **technology** keyword to the **endtechnology** line.

```
technology designkit1

model diodemodel diode is=1e-16
model diodemodel1 diode is=2e-16

endtechnology designkit1

technology designkit2

model diodemodel diode is=1e-15
model diodemodel2 diode is=2e-15

endtechnology designkit2

technology designkit1

d1 (1 0) diodemodel // generates a diode with is=1e-16

endtechnology designkit1

technology designkit2

d2 (2 0) diodemodel // generates a diode with is=1e-15

d3 (3 0) diodemodel1 //generates an error

endtechnology designkit2
```

The technology is persistent. All the sections are linked as if they form one block. The value of a variable is constant throughout the whole technology, regardless of the location where it is defined.

```
technology designkit1
is1=1e-16
endtechnology designkit1

technology designkit2
is1=2e-16
model diodemodel diode is=is2
endtechnology designkit2

technology designkit1
model diodemodel diode is=is1
d1 (1 0) diodemodel // generates a diode with is=1e-16
endtechnology designkit1
```

The technology inherits the namespace from the toplevel in a similar way as a subcircuit does. The difference is that it shares namespace of the instances and the terminals.

This example illustrates a possible use of technology. With the exception of the separation of environment variables this can be done by placing everything in subcircuits. The technologies significantly reduce the number of hierarchies which are created.

```
technology diode_designkit
#include diode_options //sets environment parameters and global variables
#include diodelibrary //contains diode_model
endtechnology diode_designkit

technology mosfet_designkit
#include mosfet_options //sets environment parameters and global variables
#include mosfetlibrary //contains mosfet_model
endtechnology mosfet_designkit

subckt mosdiode (drain gate source bulk)
parameters l=1u w=1u

technology mosfet_designkit
mosfet (drain gate source bulk) mosfet_model l=l w=w
endtechnology mosfet_designkit

technology diode_designkit
bd_diode (bulk drain) diode_model l=l w=w
bs_diode (bulk source) diode_model l=l w=w
endtechnology diode_designkit

end mosdiode
```

alternative with subcircuits:

```
subckt diode_subcircuit (anode cathode)
parameters l=1u w=1u
#include diode_options //sets global variables but not environment parameters
#include diodelibrary //contains diode_model
diode (anode cathode) l=l w=w
end diode_subcircuit

subckt mosfet_subcircuit (drain gate source bulk)
parameters l=1u w=1u
#include mosfet_options //sets global variables but not environment parameters
#include mosfetlibrary //contains mosfet_model
mosfet (drain gate source bulk) mosfet_model l=l w=w
endtechfile mosfet_designkit

subckt mosdiode (drain gate source bulk)
parameters l=1u w=1u

mosfet (drain gate source bulk) mosfet_subcircuit l=l w=w
bd_diode (bulk drain) diode_subcircuit l=l w=w
bs_diode (bulk source) diode_subcircuit l=l w=w
```

```
end mosdiode
```

The technologies allow the user a flexibility which they lose with subcircuits. One can change model or instance parameters from the instance line without including them in the subcircuit definition.

```
technology diode_designkit
#include diode_options //sets environment parameters and global variables
#include diodelibrary //contains diode_model
endtechnology diode_designkit

technology mosfet_designkit
#include mosfet_options //sets environment parameters and global variables
#include mosfetlibrary //contains mosfet_model
endtechnology mosfet_designkit

subckt mosdiode (drain gate source bulk)
parameters l=1u w=1u

technology mosfet_designkit
mosfet (drain gate source bulk) mosfet_model l=l w=w vto=0.6
endtechnology mosfet_designkit

technology diode_designkit
bd_diode (bulk drain) diode_model l=l w=w
bs_diode (bulk source) diode_model l=l w=w
endtechnology diode_designkit

end mosdiode
```

This shall also allow users to use different TMI2 directories.

4.2.6 Global nodes

Global nodes can be defined by the user. These terminals can be referred to at any level of the hierarchy by its global name. They can be defined at the top level of the design by:

//From A7.3.7

global ::= **global** <terminal_name> {<terminal_name>}

---

global is a keyword which defines the global nodes
terminal_name are legal names of terminals

In the following example, throughout different levels of hierarchy in the design, vdd! and Ground will refer to the same two terminals.

```
global vdd! Ground
```

The terminal name 0 is reserved for the ground terminal and is global. This terminal is fixed at zero voltage during simulation.

To assign an alias to the ground node, the **set_ground** keyword can be used. The terminal names which are defined through this will be treated as aliases to ground "0", indistinguishable from each other after parsing.

//From A7.3.7

set_ground ::= **set_ground** <terminal_name> {<terminal_name>}

---

set_ground is a keyword which defines the global nodes
terminal_name are legal names of terminals

In the following example, throughout different levels of hierarchy in the design, ground, gnd and "0" will refer to the same ground node.

```
set_ground ground gnd
```

4.2.7 Paramset functionality

Similar to Verilog-A, allow a set of parameters to be declared. These parameter sets are not necessarily linked to a model or subcircuit.

//From A7.5.6

paramset ::= **paramset** <name_of_paramset> < paramset_type_or_paramset> **\n**
"{" \n
{<parameter_list> **\n**}
"}" \n
name_of_paramset ::= <name>
paramset_type_or_paramset ::= **paramset** | <paramset>

---

paramset is a keyword which defines a paramset
name_of_paramset is a legal name of a paramset.
paramset_type_or_paramset is either the name of a paramset which has been declared elsewhere or the keyword paramset if this is a freshly declared paramset.
param1, value1, param2 and value2 are legal parameter names and expressions

```
paramset simulationoptions paramset
{
   gmin=1e-12 abstol=1e-8
}

parameter2 = simulationoptions.gmin

assigns value of 1e-12 to parameter2.

paramset extendedsimulationoptions simulationoptions
{
  reltol=1e-3
  abstol=1e-7
}
```

```
extendedsimulationoptions will contain 3 parameters:
gmin=1e-12
reltol=1e-3
abstol=1e-7
```

4.2.8 Conditional instantation

The user can conditionally instantiate parts of the netlist using if/then/else statements. The condition has to be resolvable by the parser during netlist parsing and flattening. It can depend on variables, user-defined function, but not on anything which can only be resolved during simulation (e.g. voltages or currents in the circuit). In contrast with the preprocessor command #if (4.1.18), the conditional expression has access to netlist variables and namespace.

//From A7.5.7

conditional_instantiation ::= **if "("** <boolean_expression> **")" \n**
"{" \n
{ <statement> }
"}" \n
[**else \n**
"{"
{ <statement> }
"}" \n]

---

if is a keyword, the boolean_expression is an expression which has a Boolean value, and which can be evaluated by the parser. Everything between the curly braces "{" and "}" will be instantiated if the value of this expression is true. Otherwise, it will be ignored.

else is a keyword, everything between the curly braces "{" and "}" will be instantiated if the value of boolean_expression is false.

```
subckt someThing (node1 node2)
parameters arg1=10 arg2=3
  if (arg2 == 3) {
     R1 (node1 node2) resistor R=arg1
  } else {
     C1 (node1 node2) capacitor C=arg1
  }
end someThing
```

4.2.9 Environment parameters (Simulator options)

Simulator options or *environment parameter* are typically common to many models. They affect model equations but are not listed as one of the model parameters. Appendix 3 contains the accepted simulator options and their default values when they are undefined. They can be assigned a value at any point in the netlist, they do not have to be defined and their names are reserved keywords. A warning message shall be sent, when a global option is redefined, the first encountered value shall be used. As an exception, a technology can have its own local global options or nodes.

//From A7.5.8

environment_parameter ::= <environment_parameter_name> **"="** <expression>
environment_parameter_name ::= **scale | temperature | global_seed | truncate**

environment_parameter_name is the name of an option, the list is in Appendix 3
expression is a numerical expression, which can be evaluated by the parser

In the following example, temperature is 25 and a warning message shall have been sent about the redefinition of temperature.

```
temperature=25
…
model…
instance…
…
temperature=30
…
subckt my_res (node0 node1)
  rpw1 (node0 node1) resistor r=100*1e-3*(temperature-25)
end my_res
```

And in the following case, temperature is here whatever value the environment parameter temperature has.

```
subckt my_res (node0 node1)
  rpw1 (node0 node1) resistor r=100*1e-3*(temperature-25)
end my_res
```

Tnom has been a global option in most netlist languages. This is not a good practice, since it is better for the modeling engineer to define tnom on the model level. This is why tnom has been removed as a standardsimulator option. It can still be used as a non-standard environment parameter, as described in the next section.

Global_seed and truncate are two global options which are used in statistical analysis. Their usage will be explained in section 4.2.10.

4.2.9.1 Non-standard environment parameters

The user can provide other simulator-specific environment parameters by using the reserved paramset **simulator_options**. If the simulator does not recognize the specific option it can be ignored by the parser.

//From A7.5.8
simulator_options ::= **simulator_options "."** <option_name> **\n**
<option_name> ::= <string>

It does not seem to be a good idea to include simulator-specific options in a design kit, but the functionality must exist.

The following statement allows the user to set the simulator option gmin to 1e-14 (if the simulator supports this option). If it does not recognize such option, the parser can ignore the statement.

```
simulator_options.gmin = 1e-14
```

4.2.10 Statistical analysis

The CMC language supports a method to add statistical variation to the value of a parameter or variable. The language uses a syntax close to the Verilog-A specification. The syntax is largely based on the one in section 9.13.2 of the Verilog-AMS LRM2.3 [1]. The C-code which describes the function can be found in Section 17.9.3 of IEEE std 1364-2005 Verilog HDL.

The following functions are supported:
For all these functions seed is a number with an integer value and is used to give consistent results for debugging. If the seed remains constant, the random numbers shall not change from run-to-run. If the seed parameter is omitted then the simulator picks a seed which is different than that of any other distribution in the netlist. The environment parameter global_seed can be defined to make the process of selecting the undefined seeds, consistent from run-to-run. If the environment parameter global_seed is not set, a random seed is assigned which shall vary from run-to-run.

The type is a string which is either "global" or "instance". If it is "global", the value is assigned once per Monte Carlo run, but when it is "instance" then a new random value is generated for each instance which uses this value and for every Monte Carlo run. This value can be omitted and it will default to global in this case.

//From A7.3.3.2
statistical_functions ::= rdist_uniform | rdist_normal | rdist_lognormal | rdist_exponential | rdist_poisson | rdist_chi_square | rdist_t | rdist_erlang | arandom | rselect
rdist_uniform ::= **rdist_uniform "("** [<seed>] **","** <start> **","** <end_real> [**","** <type>] **")"**
seed ::= <integer_number>
start_real ::= <real_expression>
end_real ::= <real_expression>
type ::= **global** | **instance**

Uniform distribution, start_real has to be smaller than end_real or an error message will be sent. The mean value will be its center (0.5*(start+end)). This function is called uniform in IEEE std 1364-2005.

//From A7.3.3.2
rdist_normal ::= **rdist_normal "("** [<seed>] **","** <mean> **","** <standard_deviation> [**","** <type> [**","** <truncate>]] **")"**
mean ::= <real_expression>
standard_deviation ::= <real_expression>
truncate ::= <real_number>

Gaussian distribution, defined by its mean and standard_deviation (real numbers). The mean value will be the value of mean. The parameter truncate is used to define the maximum number of standard deviations a value can deviate from the mean. If this parameter is omitted, the value of the environment parameter truncate is used. If the truncate parameter is present, the type parameter has to be defined. This function is called normal in IEEE std 1364-2005.

//From A7.3.3.2

---

rdist_lognormal ::= **rdist_lognormal "("** [<seed>] **","** <mean> **","** <standard_deviation> ["," <type>] **")"**

Lognormal distribution (or Galton distribution), defined by its mean and standard_deviation (real numbers). The normal logarithm of this distribution will be normally distributed. The mean value will be the value of mean. This function is not present in IEEE std 1364-2005. The probability density of this distribution is given by:

f(x)=1/(x*standard_deviation*sqrt(2*pi))*exp(-(lnx-mean)^2/(2*standard_deviation^2))

//From A7.3.3.2

rdist_exponential ::= **rdist_exponential "("** [<seed>] **","** <mean> [**","** <type>] **")"**

Exponential distribution, defined by its mean, which shall be a real number larger than 0. The mean value will be the value of mean. This function is called exponential in IEEE std 1364-2005.

//From A7.3.3.2

rdist_poisson ::= **rdist_poisson "("** [<seed>] **","** <mean> [**","** <type>] **")"**

Poisson distribution, defined by its mean. The mean value will be the value of mean, which shall be a real number larger than 0. This function is called poisson in IEEE std 1364-2005.

//From A7.3.3.2

rdist_chi_square ::= **rdist_chi_square "("** [<seed>] **","** <degrees_of_freedom> [**","** <type>] **")"**
degrees_of_freedom ::= <real_expression>

Chi square distribution, defined by its degrees_of_freedom, which shall be a real number larger than 0. The mean value will be the value of the degrees of freedom. This function is called chi_square in IEEE std 1364-2005.

//From A7.3.3.1

rdist_t ::= **rdist_t "("** [<seed>] **","** <degrees_of_freedom> [**","** <type>] **")"**

Student's T-distribution, defined by its degrees_of_freedom, which shall be a real number larger than 0. The mean value will be 0 (though it technically is undefined if degrees of freedom are smaller than or equal to 1). This function is called t in IEEE std 1364-2005.

//From A7.3.3.2

rdist_erlang ::= **rdist_erlang "("** [<seed>] **","** <k_stage> **","** <mean> [**","** <type>] **")"**
k_stage ::= <real_expression>

Erlang distribution, defined by its k_stage and mean, which shall be real numbers larger than 0. The mean value will be the value of mean. This function is called erlang in IEEE std 1364-2005.

If statistical analysis is not supported by the analysis which is performed or is disabled by the user, then the function will return the mean value of the function.

```
tox_mmglobal = rdist_normal(0,0,0.02,"global")
tox_mmslope = rdist_normal(1,0,5e-3,"instance")
model lvnfet nmos
+tox = 5.0e-009*tox_mmglobal + 5.00e-9*tox_mmslope/
sqrt(sqrt(E(l)*E(w)*E(mfactor))
```

```
global_seed=1
tox_mmglobal = rdist_normal(,0,0.02,"global") //Parser uses global_seed to set seed
tox_mmslope = rdist_normal(,0,5e-3,"instance")
model lvnfet nmos
+tox = 5.0e-009*tox_mmglobal + 5.00e-9*tox_mmslope/
sqrt(sqrt(E(l)*E(w)*E(mfactor))
```

//From A7.3.3.2

arandom ::= **arandom "("** [<seed>] [**","** <type>] **")"**

---

returns a real number with random 32-bit integer value which can be positive or negative, similar to the Verilog-A $arandom function. The same rules for seed and type apply as for the other distributions.

//From A7.3.3.2

rselect ::= **rselect "("** [<seed>] **","** <var1> **","** <var2> [**","** <type>] **")"**
var1 ::= <expression>
var2 ::= <expression>

---

randomly selects either the value of var1 or var2. Var1 and var2 can be dependent on an underlying distribution. The type of rselect only governs the choice between var1 and var2 and not the type of any underlying distributions on which they might depend.

The right hand side is equivalent to the left hand side in the following example:

`x = rselect(seed,var1,var2,type)`	`random_number = random(seed,type)`
	`selector = random_number -`
	`+2.0*int(random_number/2)`
	`x=selector*var1+(1-selector)*var2`

4.2.10.1 Correlations

To implement correlated distributions it is recommended to write out the distributions as combinations of underlying distributions.

As is shown in the following example, it is possible to create two parameters that are correlated by writing them out as combinations of three uncorrelated distributions. Care has to be taken about the standard deviation of the resulting distribution and the correlation coefficients. The correlation between these two parameters is 0.4 and the standard deviation of each is 1.

```
x=rdist_normal ( 0 , 0, 1, "instance" )
y=rdist_normal ( 1 , 0, 1, "instance" )
z=rdist_normal ( 2 , 0, 1, "instance" )
```

```
parameter1 = 0.4*x+0.9165*y //stdev= sqrt(0.4*0.4+0.9165*0.9165)=1
parameter2 = 0.4*x+0.9165*z
```

As is shown in the following example, the same can be accomplished with 3 parameters. Create three parameters with a mean of 0 and a standard deviation of 1, parameter1, parameter2 and parameter3 where parameter1 and parameter2 have a correlation coefficient of 0.2, parameter2 and parameter3 have a correlation coefficient of 0.1 and parameter1 and parameter3 have a correlation coefficient of 0.3:

```
x=rdist_normal ( 0 , 0, 1, "instance" )
y=rdist_normal ( 1 , 0, 1, "instance" )
z=rdist_normal ( 2 , 0, 1, "instance" )
w=rdist_normal ( 3 , 0, 1, "instance" )
v=rdist_normal ( 4 , 0, 1, "instance" )
u=rdist_normal ( 5 , 0, 1, "instance" )
parameter1 = 0.2*x+0.3*y+0.93274*w //stdev= sqrt(0.2^2+0.3^2+0.93274^2)=1
parameter2 = 0.2*x+0.1*z+0.97468*v //stdev= sqrt(0.2^2+0.3^2+0.97468^2)=1
parameter2 = 0.1*z+0.3*y+0.94868*u //stdev= sqrt(0.2^2+0.3^2+0.94868^2)=1
```

4.3 Hierarchy, scoping and referencing

Subcircuits can be nested, which means that subcircuits can contain other subcircuits. Variables, parameters, model cards, functions, terminals, subcircuits only exist within and below the scope of the subcircuit in which they are defined.

4.3.1 Referencing outside of scope

There are methods of referencing a subcircuit, terminal, device or model card which has been defined outside of its scope. It is only possible to reference elements within the current or at a lower level of hierarchy.

4.3.1.1 Terminal

The CMC standard language allows one to reference a terminal terminal_name in another subcircuit subckt_name, through the following call:

//From A7.3.7

terminal_reference ::= [<subckt_name> **"."**] <terminal_name>

---

subckt_name is a legal name of a subcircuit within the current scope.
terminal_name is a legal name of a node defined within the scope of subckt_name.

```
Cp1 (someNode CMP2.X1.node2) capacitor C=1.5pF
```

4.3.1.2 Instance

The CMC standard language allows one to reference an instance instance_name in a subcircuit subckt_name, through the following call:

//From A7.5.1

---

instance_reference ::= [<subckt_name> **"."**] <instance_name>

subckt_name is a legal name of a subcircuit within the current scope.
instance_name is a legal name of an instance defined within the scope of subckt_name.

```
mirror (watchI 0) ccvs Probe=CMP2.X1.R2
```

The circuit below has two distinct nodes: someNode and CMP2.X1.node2 and two components: CMP2.X1.R1 and CMP2.X1.R2.

```
subckt mySubCircuit (node1)
  R1 (node1 node2) resistor r=50
  R2 (node2 0) resistor r=50
end mySubCircuit

subckt AnotherOne (node1)
  X1 (node1) mySubcircuit
end AnotherOne

CMP2 (someNode) AnotherOne
```

4.3.1.3 Alias functionality

The CMC language allows the user to create an alias for a subcircuit or an instance, so it can be referred to by this alias instead of its longer name. This is a direct text-based replacement.

//From A7.5.3

alias ::= **alias** <alias_name> **"="** <name>
alias_name ::= <name>

alias is a keyword which allows the user to set an alias for a valid instance or subcircuit within the scope.
alias_name is the name of the alias which can be used to refer to name with in the current scope.
name is the name of an instance or subcircuit within the current scope

The following statement allows x1.i1.amp1.n1 can now be referred to as amplifier.n1

```
alias amplifier=x1.i1.amp1
```

4.3.2 Hierarchical variables

Variables can be defined hierarchically. Definitions inside a subcircuit hide global definitions with the same name. Variables defined at a higher level can be referenced inside a subcircuit. The multiple levels of hierarchy apply for subcircuits within subcircuits.

When a subcircuit is created, it inherits the namespace from the hierarchy in which it is created. If this hierarchy was a subcircuit, the parameters will be treated as if they were variables. But all variables and their values are now a copy local to the subcircuit. By changing the variable within the subcircuit, one

does not change the variable within the hierarchy in which the subcircuit was instantiated, or any higher level. It overwrites the value it inherited from a hierarchical level.
If the instance line which generates the subcircuit, contains parameter assignments these will be used to overwrite the parameter assignment within the scope of the subcircuit. Parameters are defined by the keyword **parameters**, according to the rules in 4.2.3. A variable has one value for a given level of hierarchy.

```
//Global Nodes
global vdd! Ground
//end Global Nodes

//Technology Parameters
tox=value1 vth0=value2
tparam=v1
//end Technology Parameters


subckt subckt_name (node1 node2 …)
//Subcircuit Instance Parameters
parameters mismatch1=mvalue1 mismatch2=mvalue2
parameters tparam=v3
parameter1=x parameter2=y tparam=v2
//end Subcircuit Instance Parameters
//Variables
… components here …
//end Subcircuit Parameters

//Model Instance Parameters
modelcall (node1 node2 node3 node4) nmos l=0.2u w=0.3u
//end Model Instance Parameters

end subckt_name


model nmos device_type
//Model Parameters
+ eta0=1.2    ngate=0.1
+ nfactor=2.3   ku0=2
+ toxe=tparam
//end Model Parameters
// Simulator Parameters (Switches)
+ updatelevel=1
// end Simulator Parameters
```

```
var1=1.5
var2=3.5e-6

subckt AnotherThing (node1)
  var1=2.5  // local definition of var1 hides global
  R1 (node1 0) resistor R=var1  // this is a 2.5 Ohm resistor
  C1 (node1 0) capacitor C=var2  // this is a 3.5 uF capacitor
end AnotherThing
```

```
x=3
```

```
subckt subckt1 (node1)
  x=4 // local definition of x hides global
  subckt subckt2 (node1)
    R1 (node1 0) resistor R=x
  end subckt2
end subckt1

subckt subckt3 (node1)
  parameters x=6
  R1 (node1) subckt1.subckt2 //generates a resistor with R=4
end subckt3
```

In the following example, sub1.r1=333, sup1.r2=2, sub2.xarf.r1=42,sub2.xarf.r2=43, sub3.xarf.r1=42,sub3.xarf.r2=31

```
x=3

subckt subckt1 (node1)
  parameters x=5
  R1 (node1 0) resistor r=x
end subckt1

R1 (node1) subckt1 x=7 //generates a resistor with R=7
R2 (node1) subckt1//generates a resistor with R=5

p1=3 pp1=1 p2=2
sub1 s1 p4=8
sub2 s2 p2=43
sub3 s2 p2=31
subckt s1
   parameters p1=333 p3=5 p4=p1+p3 p5=p1+p3
   r1 (1 0) resistor r=p1
   r2 (1 0) resistor r=p2
end s1
subckt s2
   parameters p2=17 pp1=42
   xarf s1 p1=pp1
end s2
```

4.3.3 Scoping of variables

It is possible to define variables with the same name at different levels of hierarchy. A variable is valid within its own scope and is inherited by any subcircuit instantiated within that scope, unless a variable with the same name is defined in this lower scope.

```
//Global Variables
tox=value1 vth0=value2
tparam=v1 ------------------------------------SCOPE LEVEL 1
//end Global Variables

subckt subckt_name (node1 node2 …)
parameter1=x parameter2=y tparam=v2 -------------SCOPE LEVEL 2

//tparam=v3  WOULD FLAG an error
… components here …
```

```
modelcall (node1 node2 node3 node4) nmos l=0.2u w=0.3u
end subckt_name

model nmos device_type
+ eta0=1.2    ngate=0.1
+ nfactor=2.3   ku0=2
+ toxe=tparam //TOXE=v1
```

```
//Global Variables
tox=value1 vth0=value2
tparam=v1 ---------------------------------------SCOPE LEVEL 1
//end Global Variables

subckt subckt_name (node1 node2 …)
parameter1=x parameter2=y tparam=v2 --------------SCOPE LEVEL 2

… components here …

modelcall (node1 node2 node3 node4) nmos l=0.2u w=0.3u

model nmos device_type
+ eta0=1.2    ngate=0.1
+ nfactor=2.3   ku0=2
+ toxe=tparam //TOXE=v2 or whatever instance parameter is passed onto subckt when it
is called from the instance line

end subckt_name
```

4.3.4 Parallel devices

The CMC language allows the user to place parallel devices in the netlist on one instance line, using the multiplicity factor, mfactor. Every instance has a parameter mfactor associated with it. If no mfactor is specified it is 1, but if the instance is instantiated by a subcircuit, the mfactor of all instance statements in the subcircuit are multiplied by this mfactor when flattening.

//From A7.5.1

mfactor ::= **mfactor "="** <real_expression>

The netlist on the left hand side will read to the parser like the one on the right hand side:

```
subckt res1 (in out)
 r1 (in out) resistor r=50 mfactor=10
 r2 (in out) resistor r=20
end res1
r1 (1 2) res1 mfactor=2
```

```
r1.r1 (1 2) resistor r=50 mfactor=20
r1.r2 (1 2) resistor r=20 mfactor=2
```

This mfactor can be used as a variable in any expression. It will use the mfactor of the statement it is found. If it is not an instance statement, it will treat the line as an instance statement to find the correct mfactor.

```
subckt res1 (in out)
 test=mfactor*2 //test is 4 here in r1 instantiation
```

```
 r1 (in out) resistor r=50 mfactor=10 dtemp=mfactor*1.5 //dtemp is 30 here
 r2 (in out) resistor r=20 dtemp=mfactor*1.5 //dtemp is 3 here
end res1

r1 (1 2) res1 mfactor=2
test=mfactor*2 //test is 2 here
```

4.3.5 Parameter scoping

The CMC language has different hierarchies of parameters which can be explicitly passed down to a model or a subcircuit.

4.3.5.1 Instance and model parameters

In traditional SPICE languages, there is a difference between instance and model parameters (this can improve simulation performance). In Verilog-A, model and instance parameters are all model parameters (similarly how parameters in a subcircuit work).

There is a potential benefit to being able to separate instance and model parameters. But there is a need to be able to overwrite a model parameter with an instance parameter. Similarly, there is a need to evaluate model parameters using the instance parameters of the instance which calls it.

To preserve the benefits of having instance and model parameters, instance parameters do not overwrite model parameters. The parser does not necessarily know which parameter is an instance and which is a model parameter. This functionality can be replicated easily by creating a model of a model overwriting just the parameters which are needed. When the user tries to overwrite a model parameter on the instance line, the simulator may send a warning to the output.

In the following example, resistor is a model with instance parameters r, l and w, while rsh is a model parameter, all their default values are 0. Resistor1 will be instantiated with the following values for its parameters: r=10, l=2u, w=0 and rsh=5. Resistor2 will be instantiated with the following values for its parameters: r=0, l=1u, w=2u and rsh=1.

```
Resistor1 (node1 node2) resistormodel1 r=10
model resistormodel1 resistormodel rsh=5
Resistor2 (node1 node2) resistormodel l=1u w=2u

model resistormodel resistor rsh=1 l=2u
```

The parameters on the instance line which calls a particular model card or subcircuit can be accessed within a certain model card through the following syntax:

//From A7.3.6

instance_parameter_call ::= **E "("** <parametername> **")"**

---

returns the value of the instance parameter parametername, if there is no such parameter parametername on the instance line an error is returned, where:

E is the function name which allows one to use an instance parameter within a model definition.

parametername is the name of the instance parameter, which has to be evaluated

This is equivalent to replacing the model by a subcircuit with the model embedded. The advantage of this approach is that the subcircuit approach requires all instance parameters, even those not used in the calculation to be defined in the subcircuit.

In the following example both netlists are equivalent:

```
instance1 (node1 node2) model_name
+y=value1
model model_name model_type x=E(y)
```

```
instance1 (node1 node2) model_name
+y=value1
subckt modelname (node1 node2)
parameters y
instance (node1 node2) model_name_sub y=y
model model_name_sub model_type x=y
end modelname
```

In the following example, E(l) means to evaluate the instance parameter called "l" on every single element of this model type, where E(myw) means to evaluate the instance parameter called "myw" on every single element of this model type and E(mfactor) means to evaluate the mfactor on every single element of this model type (see section 4.3.4). Both netlist are equivalent, except that on the left hand side no instance parameters have to be defined in the subcircuit definition

```
model polyres resistor
+rsh=80*0.40/(sqrt(E(l)*E(myw)*E(mfactor))
```

```
subckt polyres (node1 node2)
parameters l myw
r1 (node1 node2)  model_name l=l myw=myw
model model_name resistor
+rsh=80*0.40*/sqrt(l*myw*mfactor)
end polyres
```

4.3.5.2 Renaming parameters

The **rename** function can be used in a model card to rename the name of an instance or model parameter. This function is performed before any other expressions are evaluated and once a parameter has been renamed, the original name does not exist in the model card or on the instance line. When the parameter which needs to be renamed does not exist, this function is ignored. If the parameter which has to be renamed is an instance parameter, this can be identified by using the E() function. The rename functions are executed in the order they are found.

//From A7.3.6

rename_function::= **rename** "**(**" <instance_parameter_call> | <parameter_name> "**,**" <parameter_name> "**)**"

The name of the first argument will be replaced by the name given by the second argument in the function call, when the model and instance parameters are passed to the device.

In the following example both netlists are equivalent

```
model nch bsim4 rename(E(length), l)
```

```
model nch bsim4 vth0=0.66
```

`+rename(vth, vth0) vth=0.66` `M1 (d g s b) nch length=10u width=5u`	M1 (d g s b) nch l=10u width=5u

`model nch bsim4 rename(E(length), l)`
`+rename(E(width), w) rename(E(width), w2)`
`+vth0=0.66`
`M1 (d g s b) nch length=10u width=5u`
	model nch bsim4 vth0=0.66
	M1 (d g s b) nch l=10u w=5u

While the following example leads to an error sent to the output, as E(width) does not exist anymore

```
model nch bsim4 RENAME(E(length), l) RENAME(E(width), w) vth0=0.66+E(width)*0.0001
M1 (d g s b) nch length=10u width=5u
```

This can be used to remap a model card to a new version or model:

```
model bjtmodel hicuml2 cjci0=0.01u cjei0=0.01u
```

could be remapped to

```
model bjtmodel mextram504_remap
model mextram504_remap mextram504 rename(cjci0, cjc) rename(cjei0, cje)
```

4.3.6 Signal Access Functions

A voltage at a terminal can be passed to an instance or a subcircuit by passing the terminal name to the TERMINAL LIST. This can be done following the rules set in 4.3.1.2.
A current through a branch can be passed to an instance or a subcircuit by passing the name of the corresponding voltage source, current probe, linear resistor or inductor to the instance as a parameter. The instance is referenced following the rules of 4.3.1.3.

```
resistor1 (node1 node2) r=1
cccs1 (node2 0) cccs probe=resistor1 //passes the current through resistor1 to cccs1
vcvs1 (node1 0 node1 node2) vcvs //passes the voltages at nodes node1 and node2 to
vcvs1
```

4.4 Model definitions

Several models have not been standardized by the CMC, yet are part of design kits. Following is a description of models in Verilog-A or an equation/parameter list. We need to define at least inductor, mutual inductor and vcvs, vccs, cccs, ccvs with standard gain, polynomial dependence, piecewise linear and with a delay, we may also define the MOSFET level 1, the Gummel-Poon BJT and other models which are still used, but have no CMC standard. This work will be postponed until after the first version of the language has been standardized.

5 Future Work

After the language has been specified, the committee will work on other issues regarding netlists. One of these issues will be to standardize .measure statements. If it can be demonstrated that such statements are used within design kits, we will add them to the current work. Examples of this statement and its intended behavior are welcomed.

Several models have not been standardized by the CMC, yet are part of design kits. Following is a description of models in Verilog-A or an equation/parameter list. We need to define at least inductor, mutual inductor and vcvs, vccs, cccs, ccvs with standard gain, polynomial dependence, piecewise linear and with a delay, we will also define the MOSFET level 1, the Gummel-Poon BJT and other models which are still used, but have no CMC standard. This work will be postponed until after the first version of the language has been standardized.
The language may define all the output capacitances and resistors defined by model developers like PSP, BSIM and HSIM as agreed to by CMC. We may write a standardized definition for resistance and output capacitance given certain types of analysis, and a mechanism to access them.
A QA suite may be made to test the implementation of the language features in different simulators. This may be released together with the standard.
The standard may consider adding directives which could be used to identify which parts of the netlists shall be encrypted by the encryption tool.
The elaboration of the language will be defined after the first version of the language has been approved for standardization. The process of defining this elaboration will happen within the committee in a way the parser developers can design the parsers while they contribute to the elaboration scheme.
The standard may consider adding functionality to import a C-function to a netlist as a user-defined function. The committee may consider doing this both as a stand-alone compiled object and as uncompiled C-code.

6 References

[1] Accellera Verilog-AMS Language Reference Manual, version 2.3 [Online]:
http://www.accellera.org/activities/verilog-ams/VAMS-LRM-2-3.pdf

[2] Spectre User's Guide, A Frequency-Domain Simulator For Nonlinear Circuits, Program by Kenneth S. Kundert and Alberto Sangiovanni-Vincentelli, Manual by Kenneth S. Kundert and Eric Copeland, Version 1a1, August 1988

[3] Agilent Technologies, *Agilent ADS 2009A – Using Circuit Simulators*, chapter 5 "ADS Simulator Input Syntax", 2009

Appendix 1. Defined Unit Prefixes

Units Prefix	Multiplier
a	10^{-18}
f	10^{-15}
p	10^{-12}
n	10^{-9}
u	10^{-6}
m	10^{-3}
K, k	10^{3}
M	10^{6}
G	10^{9}
T	10^{12}

Appendix 2. Defined Names for Models

Name of Model	Identifier	Name of Model	Identifier
PSP	psp	diode	diode_cmc
HiSIM2	hisim	JUNCAP2	juncap2
BSIM3	bsim3	CMC Two-terminal Resistor	r2_cmc
BSIM4	bsim4	CMC Three-terminal Resistor	r3_cmc
BSIMSOI	bsimsoi	CMC MOS Varactor	mosvar_cmc
HiSIM_HV	hisim_hv	independent voltage source	vsource
HiCUM Level 0	hicuml0	independent current source	isource
HiCUM Level 2	hicuml2		
MEXTRAM	mextram	voltage-controlled voltage source	vcvs
VBIC	vbic	voltage-controlled current source	vccs
SPICE Gummel-Poon	sgp	current-controlled voltage source	ccvs
General Resistor	resistor	current-controlled current source	cccs
General Capacitor	capacitor		
General Inductor	inductor		
Mutual Inductor	mutual_inductor		

Appendix 3. Defined Simulator Parameters and Options

Name	Description
`scale`	Geometry scaling factor for instance parameters (default 1.0)
`temperature`	Circuit temperature (default 25)
`global_seed`	Seed value to base undefined seed values on (default undefined)
`truncate`	Default value of the maximum number of standard deviations a value of a normal distribution can deviate from the mean (default 6)

Appendix 4: Definitions

device: A representation of a physical device or property, such as a resistor, flux linkage, capacitor, MOSFET, bipolar transistor, etc. A device is identified by a string, the **device name**, denoting the type of device, e.g. BSIM461, VBIC95, resistor. Multiple **instance**s of a device may be created in a netlist. The **model** for the device is determined by a set of equations describing its behavior. The behavior of a device instance is determined by solution variables, **instance parameter** values, **model parameter** values, and **environment parameter** values. (Devices are often 'built-in' to the simulator, unchangeable by the user other than to change the parameters which feed into it.)

model: The equations defining terminal characteristics of an instance of a **device**, in terms of solution variables, **instance parameter** values, **model parameter** values, and **environment parameter** values.

model statement: A statement used to give **model parameter** values for the **model** which describes a device. A model statement has a name for itself (the **model name**), a **device name** (denoting the device this statement parameterizes), and a series of **model parameter** values given as keyword=value pairs.

model parameters: A list of parameters used in the model equations for a particular device. The model parameter associated with a keyword and is given a value using a keyword=value pair on a **model statement**. E.g. `phib' might be a keyword associated with the bulk potential model parameter in a MOSFET model.

instance: A particular placement of a **device** or **subcircuit**, represented in the netlist by an **instance statement.**

instance statement: A statement associated with a particular **instance**. An instance statement has a name for itself (the **instance name**), a terminal list which gives the nodes to which it is connected, a reference name, and **instance parameter**s given as keyword=value pairs. The reference name may be a **device name** in the case of a simple device using default model parameters, a **model name** referencing a **model statement** giving **model parameter** values used in the device's model, or a **subcircuit name** referring to a **subcircuit** definition.

instance parameters: A list of parameters associated with a particular instance of a device or subcircuit. If the instance is a placement of a device, the parameter is typically used in the model equations for the **device**. If the instance is a placement of a subcircuit, the parameter is typically used in expressions within the subcircuit definition. Each parameter is associated with a keyword and is given a value using a keyword=value pair on an **instance statement**.

subcircuit: A representation of a group of devices, done to allow replication of the group of devices. The subcircuit is defined by a **subcircuit definition**, and is placed in the netlist using an **instance statement**.

subcircuit definition: A subcircuit definition consists of statements beginning and ending the subcircuit definition, and containing one or more **instance statement**s, **model statements**, or **subcircuit definitions**. The statement beginning a subcircuit definition has a name for itself (the **subcircuit name**), a list of zero or more ports (which are replaced by terminals from the subcircuit instance), and an optional list of subcircuit parameters as keyword=value pairs, where the value is a default used if a value is not given on a subcircuit **instance statement**.

environment parameter: A parameter, typically common to many models, which affects model equations but is not listed as one of the model parameters. Historical examples of these would be gmin, temp, etc.

delimiter: are characters which are used to separate expressions in the language file. It can be one or more blanks or tabs, a comma or a comment. Multiple delimiters will be treated as one by the parser, except if they are a part of a string.

terminal: single node of the net, or collections of nodes (*bus terminal*)

language file/netlist: text which can be read by the parser which describes the devices, models and their topological connections for a design.

statement: a full line in the netlist, this contains all text and white space until a line break.

comment: text in the netlist which is ignored by the parser and only serves to illuminate the design to a reader.

string: ordered collection of characters, they can be used to give names to objects.

variables: symbols for values which can be used within expressions in the netlist.

parameters: are symbols for values which are used to pass a value to an instance/model/subcircuit.

bus terminal: collection of terminals, identified by a single name and their order

user-defined model: Model which has not been defined by the EDA tool, but by the user.

library: Collection of models, usually divided in sections. The sections tend to differ in the values of the variables that are used.

technology: Netlist which contains information and models, functions and variables. The technology allows the namespace of these to be separate from that of the rest of the netlist who calls it.

bus instance: collection of instances which are connected by bus terminals to the rest of the design.

parser output: text which is written out to the screen or a text file (log file) by the parser of a particular tool. The exact location of the log file (if present) or its format or content are dependent on the simulation tool.

Appendix 5: Reserved keywords

The following keywords are reserved in the CMC standard language and cannot be used as a name.

#define
#elif
#else
#endif
#if
#ifdef
#ifndef
#include
#undef
alias
abs
acos
acosh
arandom
asin
asinh
atan
atan2
atanh
ceiling
cos
dB
else
concat
cosh
E
end
encrypted
exp
explim
endencrypted
endif
endsection
endtechnology
error
false
fatal
floor
for
global
if
import
int

ln
log10
lower
max
min
model
nint
parameters
pow
range_check
rdist_chi_square
rdist_erlang
rdist_exponential
rdist_lognormal
rdist_poisson
rdist_normal
rdist_t
redefine
rename
return
rselect
section
set_ground
simulatorlanguage
sign
sin
sinh
sqrt
subckt
tan
tanh
technology
then
true
upper
warning
while

Appendix 6: Reserved macros

The following macros are reserved in the CMC standard language and cannot be defined by the user, but are defined by the simulator.

_LINUX
_MAJOR_VERSION
_MINOR_VERSION
_SOLARIS
_WINDOWS

Different EDA vendors shall define their own reserved macro which can be used to identify the simulation tool that is running. These macros shall start with "_".

Appendix 7: Formal syntax definitions

A7.1 Netlist text

statement::= <statement_line> **\n**
{ **"+"** <statement_line> **\n** | **\n** | <comment> **\n**}

statement_line::= {<printable_ASCII_character> | **\t** }

printable_ASCII_character ::= ASCII : 0x20-0x7E

comment ::= one_line_comment | multi_line_comment

one_line_comment ::= **"/" "/"**{ <printable_ASCII_character> } **\n**

multi_line_comment ::= **"/" "*"** { <multi_line_text> } **"*""/"**

multi_line_text ::= <printable_ASCII_character> | **\n**

delimiter ::= { **" "** | **\t** | **","** }

A7.2 Data-types

A7.2.1 Strings

string ::= **"**<text_of_string>**"**

text_of_string ::= {<character>}

character ::= <printable_ASCII_character> | <ASCII_code> | **\n** | **\t** | **"\" "\"** | **"\" """**

ASCII_code ::= **"\"**<octal_number><octal_number><octal_number>

octal_number ::= 0|1|2|3|4|5|6|7

select_string ::= <string_variable_name> **"(**" <i> **":**" <j> **")"**

string_variable_name := <name>

i ::= <integer_number>

j ::= <integer_number>

A7.2.2 Numbers

real_number ::= [<sign>]<unsigned_number>.<unsigned_number> |
[<sign>]<unsigned_number>[.<unsigned_number>]<exponential>[<sign>]<unsigned_number> |

[<sign>]<unsigned_number>[.<unsigned_number>]<scalefactor>

sign ::= **"+"** | **"-"**

unsigned_number ::= <decimal_number>{<decimal_number>}

decimal_number ::= 0|1|2|3|4|5|6|7|8|9

exponential ::= **e** | **E**

scalefactor ::= **a | f | p | n | u | m | k | K | M | G | T**

integer_number ::= [<sign>]<unsigned_number> | **int "("** <real_expression> **")"**

boolean_value ::= **true** | **false** | <real_number>

A7.2.3 Vectors

vector ::= real_vector | boolean_vector | string_vector

real_vector ::= "**[**" <real_expression> { "**,**" <real_expression> } "**]**"

boolean_vector ::= "**[**" **<**boolean_expression> | { "**,**" <boolean_expression> } "**]**"

string_vector ::= "**[**" **<**string_expression { "**,**" <string_expression> } "**]**"

A7.3 Language constructs

A7.3.1 Identifiers

name ::= <unquoted_name> | <quoted_name> excluding <keyword>

unquoted_name ::= <starting_name_character> { <name_character> }

quoted_name ::= **"""** <printable_ASCII_character> { <printable_ASCII_character> } **"""**

name_character ::= **a-z** | **A-Z** | **0-9** | **"_"** | **"#"** | **"!"**

starting_name_character ::= **a-z** | **A-Z** | **0-9** | **"_"** | **"!"**

A7.3.2 Operators

unary_operator ::= "**-**" | "+"

binary_operator ::= "+" | "**-**" | "*" | "/" | "=" "=" | "<" | "<" "=" | ">" | ">" "=" | "**!**" "="

string_operator ::= "=" "=" | "**!**" "="

real_ ternary_conditional_operation ::=
"(" <boolean_expression> **")"** **"?"** <real_expression> **":"** <real_expression> |
<boolean_expression> **"?"** <real_expression> **":"** <real_expression>

boolean_ternary_conditional_operation::=
"(" <boolean_expression> **")"** **"?"** <boolean_expression> **":"** <boolean_expression> |
<boolean_expression> **"?"** <boolean_expression> **":"** <boolean_expression>

string_ternary_conditional_operation::=
"(" <boolean_expression> **")"** **"?"** <string_expression> **":"** <string_expression> |
<boolean_expression> **"?"** <string_expression> **":"** <string_expression>

A7.3.3 Functions

function ::= <function_name> **"("** [<function_parameter_list>] **")"**

function_name ::= <name>

parameter ::= <real_number> | <boolean_value> | <string> | <vector> | <expression>

function_parameter_list ::= <parameter> {"," <parameter>}

function_definition ::= <function_name> **"("** [<parameter_name_list>] **")"** **"="** **\n**
"{" **\n**
[{<function_line> **\n**}]
return <expression> **\n**
"}" **\n**

function_line ::= <function_conditional> | <variable_assignment>

parameter_name_list ::= <parameter_name> { **","** <parameter_name>}

real_function_definition ::= <real_function_name> **"("** [<parameter_name_list>] **")"** **"="** **\n**
"{" **\n**
[{<function_line> **\n**}]
return <real_expression> **\n**
"}" **\n**

real_function_name ::= <name>

real_function_call ::= <real_function_name> **"("** [<function_parameter_list>] **")"**

boolean_function_definition ::= <boolean_function_name> **"("** [< function_parameter_name_list>] **")"**
"=" **\n**
"{" **\n**
[{<function_line> **\n**}]

return <boolean_expression> **\n**
"}" \n

boolean_function_name ::= <name>

boolean_function_call ::= <boolean_function_name> **"("** [**<**function_parameter_list>] **")"**

string_function_definition ::= <string_function_name> **"("** [<parameter_name_list>] **")" "=" \n**
"{" \n
[{<function_line> **\n**}]
return <string_expression> **\n**
"}" \n

string_function_name ::= <name>

string_function_call ::= <string_function_name> **"("** [**<**function_parameter_list>] **")"**

function_conditional ::= **if "("** <boolean_expression> **")" \n**
"{" \n
{<function_line> **\n**}
"}" \n
[**else \n**
"{" \n
{<function_line> **\n** }
"}" \n]

A7.3.3.1 Defined functions

natural_logarithm_function ::= **ln "("** <x> **")"**

log10_function ::= **log10 "("** <x> **")"**

exponential_function ::= **exp "**(" x **")"**

limiting_exponential_function ::= **explim "("** <x> **","** <y> **")"**

power_function ::= **pow "(" <**x> **","** <y> **")"**

sine_function ::= **sin "("** <x> **")"**

cosine_function ::= **cos "(" <**x> **")"**

tangent_function ::= **tan "(" <**x> **")"** returns

inverse_sine_function ::= **asin "("** <x> **")"**

inverse_cosine_function ::= **acos "("** <x>

inverse_tangent_function ::= **atan "("** <x> **")"**

atan2_function ::= **atan2 "(" <**x> **","** <y> **")"**

hyperbolic_sine_function ::= **sinh "("** <x> **")"**

hyperbolic_cosine_function ::= **cosh "("** <x> **")"**

hyperbolic_tangent_function ::= **tanh "(" <**x> **")"**

inverse_hyperbolic_sine_function ::= **asinh "("** <x> **")"**

inverse_hyperbolic_cosine_function ::=**acosh "(" <**x> **")"**

inverse_hyperbolic_tangent_function ::= **atanh "("** <x> **")"**

absolute_value_function ::= **abs "(" <**x> **")"**

square_root_function ::= **sqrt "(" <**x> **")"**

db_function ::= **dB "("** <x> **")"**

integer_value_function ::= **int "("** <x> **")"**

floor_function ::= **floor "(" <**x> **")"**

ceiling_function ::= **ceiling "("** <x> **")"**

nearest_integer_function ::= **nint "(" <**x> ")"

sign_function ::= **sign"("** <x> **")"**

minimum_function ::= **min "(" <**x> **","** <y> **")"**

maximum_function ::= **max "("** <x> **","** <y> **")"**

x::= <real_expression>

y::= <real_expression>

real_predefined_function ::= natural_logarithm_function | log10_function | exponential_function | limiting_exponential_function | power_function | sine_function | cosine_function | tangent_function | inverse_sine_function | inverse_cosine_function | inverse_tangent_function | atan2_function | hyperbolic_sine_function | hyperbolic_cosine_function | hyperbolic_tangent_function | inverse_hyperbolic_sine_function | inverse_hyperbolic_cosine_function | inverse_hyperbolic_tangent_function | square_root_function | db_function |

integer_value_function | floor_function | ceiling_function | nearest_integer_function | sign_function | minimum_function | maximum_function | absolute_value_function | range_check

uppercase_function ::= **upper "("** <s> **")"**

lowercase_function ::= **lower "("** <s> **")"**

concatenate_function ::= **concat "("** <s> **","** <t> **")"**

s::= <string_expression>

t::= <string_expression>

A7.3.3.2 Statistical functions

statistical_functions ::= rdist_uniform | rdist_normal | rdist_lognormal | rdist_exponential | rdist_poisson | rdist_chi_square | rdist_t | rdist_erlang | arandom | rselect

rdist_uniform ::= **rdist_uniform "("** [<seed>] **","** <start> **","** <end_real> [**","** <type>] **")"**

seed ::= <integer_number>

start_real ::= <real_expression>

end_real ::= <real_expression>

type ::= **global** | **instance**

rdist_normal ::= **rdist_normal "("** [<seed>] **","** <mean> **","** <standard_deviation> [**","** <type> [**","** <truncate>]] **")"**

mean ::= <real_expression>

standard_deviation ::= <real_expression>

truncate ::= <real_number>

rdist_lognormal ::= **rdist_lognormal "("** [<seed>] **","** <mean> **","** <standard_deviation> ["," <type>] **")"**

rdist_exponential ::= **rdist_exponential "("** [<seed>] **","** <mean> [**","** <type>] **")"**

rdist_poisson ::= **rdist_poisson "("** [<seed>] **","** <mean> [**","** <type>] **")"**

rdist_chi_square ::= **rdist_chi_square "("** [<seed>] **","** <degrees_of_freedom> [**","** <type>] **")"**

degrees_of_freedom ::= <real_expression>

rdist_t ::= **rdist_t "("** [<seed>] **","** <degrees_of_freedom> [**","** <type>] **")"**

rdist_erlang ::= **rdist_erlang "("** [<seed>] **","** <k_stage> **","** <mean> [**","** <type>] **")"**

k_stage ::= <real_expression>

arandom ::= **arandom "("** [<seed>] [**","** <type>] **")"**

rselect ::= **rselect "("** [<seed>] **","** <var1> **","** <var2> [**","** <type>] **")"**

var1 ::= <expression>

var2 ::= <expression>

A7.3.4 Expressions

expression ::= <real_expression> | <string_expression> | <boolean_value>

real_operation ::= [<unary_operator>] <real_expression> [<binary_operator> <real_expression>] | **"("** [<unary_operator>] <real_expression> [<binary_operator> <real_expression>] **")"**

string_operation ::= <string_expression> [<string_operator> <string_expression>] | **"("** <string_expression> [<string_operator> <string_expression>] **")"**

string_comparison ::= <string_expression> **"="** **"="** <string_expression> | <string_expression> **"!"** **"="** <string_expression>

real_expression ::= <real_number> | <real_operation> | <real_function> | <real_variable> | <real_parameter> | <string_comparison> | <real_ternary_conditional_operation>

boolean_expression ::= <real_expression> | <boolean_value> | <boolean_ternary_conditional_operation>

string_expression ::= <string> | <string_operation> | <string_function> | <string_variable> | <string_parameter> | <string_ternary_conditional_operation>

real_function ::= <real_predefined_function> | <real_function_call> | <statistical_function>

string_function ::= <string_predefined_function> | <string_function_call>

boolean_function ::= <boolean_predefined_function> | <boolean_function_call>

A7.3.5 Variables

variable_assignment ::= <single_variable_assignment> { <delimiter> <single_variable_assignment> }

single_variable_assignment ::= <real_variable_assignment> | <boolean_variable_assignment> | <string_variable_assignment> | <vector_variable_assignment>

real_variable_assignment := <real_variable> "=" <real_expression>

real_variable := <name>

boolean_variable_assignment := <boolean_variable> "=" <boolean_expression>

boolean_variable := <name>

string_variable_assignment := <string_variable> "=" <string_expression>

string_variable := <name>

vector_variable_assignment := <vector_variable> "=" <vector>

vector_variable := <name>

redefine_variable ::= redefine_real_variable | redefine_boolean_variable | redefine_string_variable | redefine_vector_variable

redefine_real_variable ::= **redefine** <real_variable> "=" <real_expression>

redefine_boolean_variable ::= **redefine** <boolean_variable> "=" <boolean_expression>

redefine_string_variable ::= **redefine** <string_variable> "=" <string_expression>

redefine_vector_variable ::= **redefine** <vector_variable> "=" <vector_expression>

A7.3.6 Parameters

parameter_list ::= <single_parameter_assignment> { <delimiter> <single_parameter_assignment> }
single_parameter_assignment ::= <real_parameter_assignment> | <boolean_parameter_assignment> | <string_parameter_assignment> | <vector_parameter_assignment>

real_parameter_assignment := <real_parameter> ["=" <real_expression>]

real_parameter := <name>

boolean_parameter_assignment := <boolean_parameter> ["=" <boolean_expression>]

boolean_parameter := <name>

string_parameter_assignment := <string_parameter> ["=" <string_expression>]

string_parameter := <name>

vector_parameter_assignment := <vector_parameter> ["=" <vector>]

vector_parameter := <name>

parameter = <real_parameter> | <boolean_parameter> | <string_parameter> | <vector_parameter>

instance_parameter_call ::= **E "("** <parametername> **")"**

rename_function::= **rename** "**(**" <instance_parameter_call> | <parameter_name> "**,**" <parameter_name> "**)**"

A7.3.7 Terminals

terminal_assignment ::= "**.**" <name_of_terminal_in_model> "**(**" <terminal_name> "**)**"

name_of_terminal_in_model ::= <name>

terminal_name ::= <name> | "**?**"**UNCONNECTED**

terminal_list ::= [<terminal_name>]{ <delimiter> <terminal_name> } | **"("** [<terminal_name>]{ <delimiter> <terminal_name> } **")"** | [<terminal_assignment>]{ <delimiter> <terminal_assignment> } | **"("** [<terminal_assignment>]{ <delimiter> <terminal_assignment> } **")"**

terminal ::= <terminal_assignment> | <terminal_name>

bus_terminal ::= <terminal_name> "[" <start_integer> ":" <stop_integer> "]"

start_integer ::= <integer_number>

stop_integer ::= <integer_number>

bus_terminal_reference ::= <terminal_name> "[" <start_integer> [":" <stop_integer>] "]"

busterminal_assignment ::= **"."** <name_of_terminal_in_model> **"("** <list_of_terminals> **")"**

list_of_terminals ::= <terminals> { <delimiter> <terminals> }

terminals ::= <terminal> | <bus_terminal>

global ::= **global** <terminal_name> {<terminal_name>}

set_ground ::= **set_ground** <terminal_name> {<terminal_name>}

terminal_reference ::= [<subckt_name> **"."**] <terminal_name>

A7.4 Netlist operations

A7.4.1 Importing

load_veriloga ::= **import """veriloga""" "," """**<file_name>**"""**

load_compiled_veriloga ::= **import """compiled_veriloga""" "," """**<file_name>**"""**

load_tmi2 ::= **import """TMI2""" "," """**<path>**"""**

path ::= <string>

A7.4.2 C-Preprocessor commands

preprocessor_define ::= **#define** <token1> [**"("** <parameter_name> { **","** <parameter_name>} **")"**]
[<token2>]

parameter_name ::= <token>

token1 ::= <identifier_token>

token2 ::= <token>

token::= <text_of_string>

identifier_token::= <identifier_token_character>{<identifier_token_character>}

identifier_token_character ::= <printable_ASCII_character> except <delimiter>

preprocessor_undefine ::= **#undef** <token1>

preprocessor_ifdef::= **#ifdef** <token1> **\n**
{ <statement> }
[**#else** { <statement> } **\n**]
#endif \n

preprocessor_ifndef::= **#ifndef** <token1> **\n**
{ <statement> }
[**#else** { <statement> } **\n**]
#endif \n

preprocessor_include ::= **#include** """ <file_name> """

preprocessor_if ::= **#if** <boolean_expression> **\n**
{ <statement> }
{**#elif** <boolean_expression> **\n**
{ <statement> }
[**#else \n**

{ <statement> }]
#endif \n

A7.4.3 Interface with other languages

set_simulatorlanguage ::= **simulatorlanguage "="** <language_name>

language_name ::= **cmc_standard** | **VerilogA** | <string>

A7.4.4 Interface with output

warning ::= **warning "("** <warning_message> [{"," <argument>}] **")"**

warning_message ::= <string>

argument ::= <expression>

range_check ::= **range_check "("** <input> **","** <lower_bound> **","** <upper_bound> **","**
<warning_message> **")"**

input ::= <real_expression>

lower_bound ::= <real_expression>

upper_bound ::= <real_expression>

error ::= **error "("** <warning_message> [{"," <argument>}] **")"**

fatal ::= **fatal "("** <warning_message> [{"," <argument>}] **")"**

A7.4.5 Interface with encryption

encryption ::= **encrypted \n**
{ <line> }
endencrypted \n

line::= { <any_character> }

any_character ::= ASCII : 0x00-0xFF

A7.5 Circuit components

A7.5.1 Instances

instance ::= <instance_name> <terminal_list> <instance_type> [<parameter_list>]

instance_name ::= <name>

instance_type := <model_name> | <subcircuit_name> | <string_parameter>

bus_instance ::= <instance_name> **"["** <start_integer> **":"** <stop_integer> **"]"** <terminal_list> <instance_type> [<parameter_list>]

bus_instance_reference ::= <instance_name> "[" <start_integer> [":" <stop_integer>] "]"

instance_reference ::= [<subckt_name> **"."**] <instance_name>

mfactor ::= **mfactor "="** <real_expression>

A7.5.2 Models

model ::= **model** <model_name> <device_type> [<parameter_list>]

model_name ::= <name>

device_type ::= <model_name> | <model_primitive> | <string_parameter>

model_primitive ::= psp | hisim | bsim3 | bsim4 | bsimsoi | hisim_hv | hicuml0 | hicuml2 | mextram504 | vbic | sgp | resistor | capacitor | inductor | mutual_inductor | diode_cmc | juncap2 | r2_cmc | r3_cmc | mosvar_cmc | vsource | isource | vcvs | vccs | ccvs | cccs | <string>

binned_model ::= **model** <model_name> <device_type> **\n**
"{" \n
<binning_label> **":"** [<parameter_list>] **\n**
{ <binning_label> **":"** [<parameter_list>] **\n** }
"}" \n

binning_label ::= <string>

binned_model_2 ::= **model** <model_name> <device_type> **\n**
"{" \n
<binning_label> **":"** <binning_condition> {<binning_condition>} [<parameter_list>] **\n**
{ <binning_label> **":"**<binning_condition> {<binning_condition>} [<parameter_list>] **\n** }
"}" \n

binning_condition ::= **<**parameter_name> **from** <enclosure_start> <real_expression> **":"** <real_expression> <enclosure_end>

enclosure_start ::= **"["** | **"("**

enclosure_end ::= **"]"** | **")"**

A7.5.3 Subcircuits

subcircuit ::= **subckt** <subckt_name> < subcircuit_terminal_list> **\n**
{**parameters** <parameter_list> **\n** }
{ <statement> }
end <subckt_name> **\n**

subckt_name ::= <name>

subcircuit_terminal_list ::= [<subcircuit_terminal_name>]
{ <delimiter> <subcircuit_terminal_name> }
| "(" [<subcircuit_terminal_name>]{ <delimiter>
<subcircuit_terminal_name> } ")"

optional_terminal ::= <terminal_name> "(" <terminal_default> ")"

subcircuit_terminal_name ::= <terminal_name> | <optional_terminal>

terminal_default ::= <previously_defined_terminal_name> | "**0**" | <global> | "**?**"**UNCONNECTED**

previously_defined_terminal_name ::= <terminal_name>

alias ::= **alias** <alias_name> **"="** <name>

alias_name ::= <name>

A7.5.4 Libraries

section ::= **section** <section_name> **\n**
{ <statement> }
endsection <section_name> **\n**

section_name ::= <name>

include_library ::= **#include """** <library_filename> **"""** **section "="** <section_name> **\n**

library_filename ::= <file_name>

A7.5.5 Technologies

technology ::= **technology** <name_of_technology> **\n**
[<content_of_technology>]
endtechnology <name_of_technology> **\n**

name_of_technology ::= <name>

content_of_technology ::= {<statement>}

A7.5.6 Paramsets

paramset ::= **paramset** <name_of_paramset> < paramset_type_or_paramset> **\n**
"{" \n
{<parameter_list> **\n**}
"}" \n

name_of_paramset ::= <name>

paramset_type_or_paramset ::= **paramset** | <paramset>

A7.5.7 Conditional instantiation

conditional_instantiation ::= **if "("** <boolean_expression> **")" \n**
"{" \n
{ <statement> }
"}" \n
[**else \n**
"{"
{ <statement> }
"}" \n]

A7.5.8 Environment Parameters

environment_parameter ::= <environment_parameter_name> **"="** <expression>

environment_parameter_name ::= **scale | temperature | global_seed | truncate**

simulator_options ::= **simulator_options "."** <option_name> **\n**

<option_name> ::= <string>