EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Implementation of a NDML++ compiler

van Gemert, Q.J.A.

Award date:
1988

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 17. May. 2025

EINDHOVEN UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING
DESIGN AUTOMATION GROUP

IMPLEMENTATION OF A NDML++ COMPILER

Q.J.A van Gemert

Master thesis
reporting on graduation work
performed from 01.11.87 to 24.08.88
by order of prof. Dr. Ing. J.A.G. Jess
and supervised by Dr. Ir. J.T.J van Eijndhoven

The Eindhoven University of Technology is not responsible
for the contents of training and thesis reports.

CONTENTS

SUMMARY 1

1. INTRODUCTION 2
   1.1 From NDML++ to NDML 2
   1.2 General overview of the compiler 2
   1.3 Portability and maintainability 2

2. THE INPUT MANAGEMENT SYSTEM 3
   2.1 The file management data structures 3
   2.2 Input management functions 4

3. THE LEXICAL ANALYZER 5
   3.1 Terminology and notations 5
   3.2 Pattern recognizers 6
   3.3 Lex, a lexical analyzer generator 8

4. THE LEXICAL ANALYZER FOR THE NMDL++ LANGUAGE 10
   4.1 The Normal mode 10
   4.2 The scanning mode 11
   4.3 The comment mode 11
   4.4 The include mode 11

5. THE PARSER 12
   5.1 The design of a language 12
   5.2 LR parsers 16
   5.3 Yacc, an LALR parser generator 17

6. THE PARSER FOR THE NDML++ LANGUAGE 19
   6.1 The parser data structures 19
   6.2 Construction of the parse tree 21
   6.3 Error recovery 22

7. THE TRANSLATOR 23
   7.1 The data structures 23
   7.2 translation of the parse tree fields 24
   7.3 Translation of the most common node types 24
   7.4 Functions and expressions 25
   7.5 Indices and dimension specifications 25
   7.6 System instances and terminal interconnect 26
   7.7 Conditional and repetitive statements 27

LIST OF FIGURES

Figure 1. Elements of the NDML++ compiler 2
Figure 2. List of defining rules 5
Figure 3. DFA recognizing (**a** | **b**)* **a b b** 6
Figure 4. NFA for regular definitions p1..pn 8
Figure 5. Schematic lexical analyzer 9
Figure 6. Parse tree for - (**i d** + **i d**) 13
Figure 7. Model of an LR parser 17
Figure 8. Parse tree for the string: '10' '+' 'i' '+' '30' 19
Figure 9. Parse tree for a list node 20

SUMMARY

This report presents the design of a compiler for the extended Network Modeling and Description Language (NDML++). The compiler is written in C, with the use of Lex and Yacc. The input management functions gather the input from text files linked by include statements, and from libraries. The lexical generator that partitions the input into tokens for the parser is generated with the use of Lex, a Lexical analyzer generator. A parser that constructs a parse tree for each NDML++ system definition, is generated with Yacc, a parser generator. The parse trees are translated with the use of two recursive routines that handle the parse tree nodes, and fields respectivily. This result in one file, containing all system defintions in the SNMDL language.

1. INTRODUCTION

Simply stated, a compiler is a program that reads a specification written in the source language, and translates it into an equivalent specification, the target language. As an important part of the translation process, the compiler reports to the user the occurrence of syntactic, semantic, or other errors.

1.1 From NDML++ to NDML

In our case the source language is NDML++, which stands for an extended version of the Network Description & Modeling Language [1],[2]. The target language is a simple version of NDML, which forms the input for the PWL (PieceWise Linear) simulator [3].

The source files can be distributed, the include mechanism (discussed in the next chapter) specifies how the different files are linked. A library can be provided, that contains specifications prepared by expert users of the language. It is also possible to read specifications stored in the ICD Database format [4], with use of the network extraction program **icd2ndml**.

All parameters valid within the NDML++ language are resolved, each model is translated separately for each different parameter set. The results of expressions are calculated, for parameter all values are available at compile time. Vector and bundle concepts are reduced to simple NDML specifications. Furthermore semantic checks are done, that are accompanied by error messages pinpointing the errors.

1.2 General overview of the compiler

Opposed to syntax directed translation, the NDML++ compiler generates an intermediate data structure, that is used in subsequent phases of the translation process. The compiler can be divided in parts as shown in figure 1, data structures are printed italic.

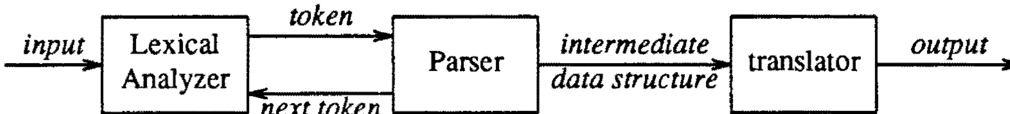


Figure 1. Elements of the NDML++ compiler

The input is assembled from text files that are gathered by include file references, libraries and the ICD database. The lexical analyzer and the parser form a team, each time the parser is ready for the next token, it calls the lexical analyzer which extracts the next token from the input stream, and returns it to the parser, then parsing continues until it needs the next token. The parser constructs the intermediate data structure, in our case a list of parse trees. The translator transforms the intermediate data structure into one simplified NDML description.

1.3 Portability and maintainability

The compiler is written in C, as defined by Kernighan and Ritchie [5]. The C code is also (cross-) checked by Lint [6],[7], the C program checker. C code that passes Lint without causing any errors and warnings, usually will not cause many problems porting. The lexical analyzer is generated by Lex [8],[9], a lexical analyzer generator, available on most UNIX and UNIX related systems. The parser is generated by Yacc [10],[11], a LALR parser generator, also available on most UNIX and UNIX related systems. Both the lexical analyzer and the parser can be updated relatively easy when the Lex and Yacc specifications are changed and compiled.

2. THE INPUT MANAGEMENT SYSTEM

The input management system handles the redirection of input files, library files, or other I/O streams, to the lexical analyzer. Input files are linked with include statements, just like this is done in the C programming language, only without the preceding '#' character. Include statements can be used everywhere in the specification file, but it is considered bad programming style to use them but only at the beginning of the file. Libraries made and maintained by experienced users can be used by setting a correct path with the UNIX environment variable PWL_PATH. The file name of a library model must end with the model name followed by "/pwl_model", "/call" or "/ndml". Currently the input for the piecewise linear simulator is generated with the use of ESCHER. These descriptions are translated by the NDML++ compiler: the input from the ICD data base is translated by **icd2ndml** and internally "piped" to the compiler.

2.1 The file management data structures

For each I/O stream that is parsed a *filenode* structure is maintained containing the following information:

- File name, this can also be a pipe name if we're reading from the ICD database.
- The file list, that contains all include files listed so far. Each include file is stored in a *filenode*. Library models are also added to this list before they are processed.
- A pointer to the module list, which contains modules that have been referenced, but which description has not been found yet. The elements in the module list are *module_element* structures, each *module_element* contains the module name and a link to the next *module_element*.
- A pointer to the parent *filenode*, this is used to find a module description that is referenced by the parent.
- A link to the next *filenode* (include file) in the list.

The names of all modules that have been parsed are kept in a linked list, the global *first_mdl_descr* points to the first element in this list. Before a module description is parsed this list is checked first, this ensures us that each description is never read more then once.

2.2 Input management functions

The most important input management functions will be discussed in this paragraph, all input management functions are located in the C files file.c and environment.c.

The C function *parsefiles()* is used to parse an I/O stream. All I/O streams are processed recursively, this means that if an include file or library model is found this is processed in the same way as all other files. First all globals, like for instance the line number of the file, and the syntax error number are set. The input is opened, either from standard input, from a text file, or from the ICD database using a pipe from **icd2ndml**. The file is parsed, and closed after completion. While there are still unresolved module descriptions in the module list, we do one of the following:

1. If there are include files left in the file list we call *parsefiles()* with the first include file in the list.
2. We try to find the module description in the library or ICD data base, this is done with *in_lib()* and *in_dbase()* respectively. If a module description is found, *addfilenode()* is called with the correct library path or pipe name. *addfilenode()* allocates and initializes a *filenode*, and links it to the file list of the current *filenode*.
3. If the module description is still not found, we issue an error message reporting the module description missing.

After all modules have been found, a warning message is issued for the include files that still are in the file list, but are not used. The filenode is removed with *rmfilenode()*, and the function returns.

mdl_used() checks if a module name occurs in the module list of the current filenode and its parent, if so it removes this element. This function is used by the lexical analyzer to decide whether to read a module description or not.

3. THE LEXICAL ANALYZER

The lexical analyzer is the first phase of the compiler. Its main task is to read the input characters and produce a sequence of tokens the parser uses for syntax analysis. Apart from this some secondary tasks can be performed. The source text is stripped from comments and white space in the form of blank, tab, and newline characters. The file names and line numbers are stored for reference within error messages that should pinpoint the errors. Also the include file mechanism can be handled here.

3.1 Terminology and notations

A Set of strings is described by a rule called a pattern. A lexeme is a sequence of characters matched by a pattern. For each lexeme that matches a pattern the same token is produced. When more than one pattern matches a lexeme, the lexical analyzer must provide additional information. The additional information is collected into attributes. The tokens influence parsing decisions, the attributes influence the translation process. In most practical cases a token has only one attribute, this can be stored into a union that can take a number of different attributes.

Example 1. Tokens, lexemes and an informal description of the pattern.

Token	Sample lexeme	Informal description of the pattern
relop	<=,==,<>	Relational operand
int	354,78,0	Integer constant
id	pi,count	Letter followed by letters and digits

An alphabet or character class Σ denotes a finite set of symbols. Typical examples of symbols are characters and letters. An alphabet may then for instance be all lower case letters from the (normal) alphabet. A string is a ordered finite set of symbols drawn from a certain alphabet (ε denotes the empty string). A string can be thought of a word in a sentence. A language denotes a set of strings over some fixed alphabet. A well formed Pascal program and the English language are two examples of a correct language. Regular expressions can be used to describe patterns.

Example 2. We can describe a Pascal identifier as

letter(letter|digit)*

The vertical bar means "or", the parentheses are used to group subexpressions, the star means "zero or more instances of" the parenthesized expression, and the juxtaposition of letter means "concatenation".

As we see a regular expression is built up out of simpler regular expressions using a set of defining rules. Figure 2 lists a number of defining rules with an informal description.

<table>
<tr><td>(r)|(s)</td><td>A string can be formed by regular expression r or s.</td></tr>
<tr><td>(r)(s)</td><td>A string can be formed by concatenation of a string formed by r to that of one formed by s.</td></tr>
<tr><td>(r)*</td><td>A string can be formed by concatenation of zero or more strings formed by r.</td></tr>
<tr><td>(r)+</td><td>A string can be formed by concatenation of one or more strings formed by r.</td></tr>
<tr><td>[a-z]</td><td>A character class: any character a and z and between.</td></tr>
</table>

Figure 2. List of defining rules

Parentheses may be omitted if we assume that * has an higher precedence than concatenation or +, and that concatenation has an higher precedence than +. For notational convenience, we may wish to give names to regular expressions. We define the use of such a name as the use of the expression that stands for this name. A regular definition is a sequence of such definitions.

Example 3. The regular definition of an unsigned number in Pascal.

> **digit** > [0-9]
> **digits** > (**digit**)+
> **optional_fraction** > '.' **digits** | ε
> **optional_exponent** > (E(+|-|ε) **digits**)|ε
> **num** > **digits optional_fraction optional_exponent**

3.2 Pattern recognizers

Now that we have the means by which we can define the strings that are recognizable by the lexical analyzer, we must find a way to translate this into a algorithm. Most commonly an algorithm called recognizer is used. A recognizer is a program that takes a input string x and answers "yes" if x matches the pattern and "no" otherwise. We compile each regular expression into a recognizer by constructing a transition diagram called a finite automaton. Figure 3 shows such an finite automaton represented by a labeled directed graph, called a transition graph. The nodes represent the states that the recognizer can be in, and the labeled edges represent the transition function for a distinct input symbol. The edge "start" marks the start state, and the nodes with a double circle indicate the accepting states. A finite automaton can be deterministic (DFA) or nondeterministic (NFA), where nondeterministic means that more than one transition out of a state may be possible for the same input symbol. In the case of figure 3 the DFA recognizes the pattern (a|b)***abb**

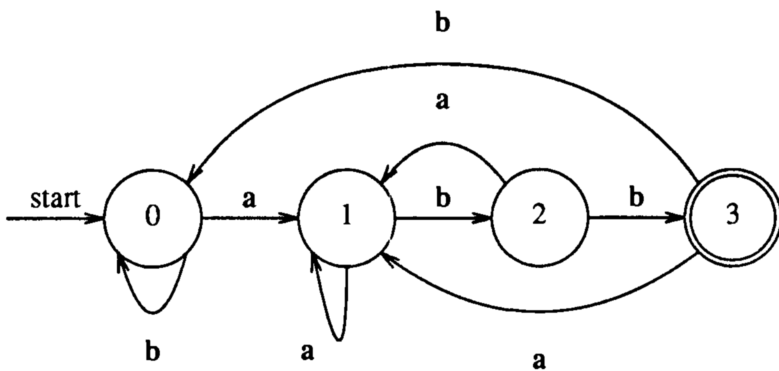
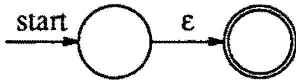


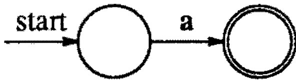
Figure 3. DFA recognizing (a|b)***abb**

Using Thompson's construction we can translate all regular expressions into an NFA:

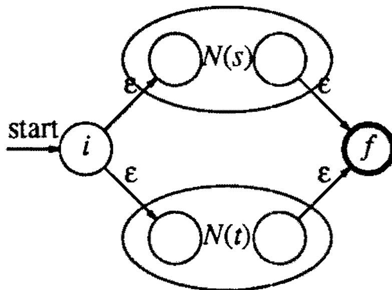
1. For ε, we construct the NFA:



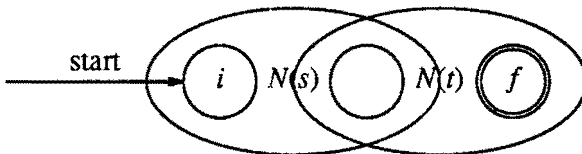
2. For a in Σ, we construct the NFA:



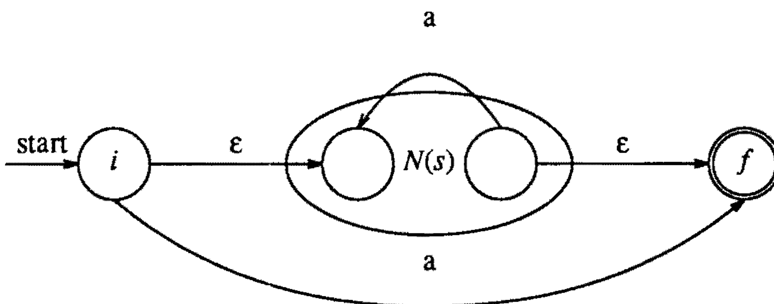
3. Suppose $N(s)$ and $N(t)$ are NFA's for regular expressions s and t.
   - For the regular expression $s|t$, we construct the following composite NFA $N(s|t)$:



   - For the regular expression st, we construct the composite NFA $N(st)$:



   - For the regular expression $s*$, we construct the composite NFA $N(s*)$:



A NFA can be converted into a DFA when each state of the DFA corresponds to a (possible) set of NFA states [12]. Although the resulting DFA has far more states, it can be advantageous to use a DFA. For a NFA more than one transition is possible on a input symbol, this makes it hard to simulate a NFA with an computer program. Furthermore DFA optimizers are able to reduce the number of states considerably.

3.3 Lex, a lexical analyzer generator

Lex is a software tool commonly found on UNIX, or UNIX related systems that constructs a lexical analyzer from a specification in the Lex language. The Lex language is an extended version of the regular definitions discussed previously. A Lex program consist of three parts, definitions, rules and user routines. Definitions are plain regular definitions that can be used in rules. Rules are regular definitions that have associated actions, if the pattern matches the rule, the action is executed. The user routines Lex merely copies to the output file. For a more extend definition of the Lex language see [8],[9].

Example 4: If the lexical analyzer matches the pattern "ID" for an identifier it returns its token and attribute(s) to the parser.

```
ID [a-zA-Z]*
%%
{ID} {
   if(not_a_keyword(yytext){
   yylval = yytext;
   return(1);
}
else
   return(keyword_value(yytext));
}
. return(0);
```

Where yylval is the variable in which attributes are passed to the parser, and not_a_keyword() and keyword_value(), are external functions that check if the identifier is a keyword, and calculate the keywords rank number respectively. The resulting lexical analyzer will return "1" and the identifiers lexeme if it matches an identifier, a keyword value if it matches a keyword, and "0" otherwise.

The recognizer that Lex generates looks for lexemes in the input buffer that match a pattern. If more than one pattern matches, the recognizer chooses the longest lexeme matched. If there are two or more patterns that match the longest lexeme, the first listed matching pattern is used. If the resulting pattern is a rule the corresponding action is executed. If p1..pn are rules for which all the definitions that are used are substituted, a NFA as in figure 4 can be constructed to match there patterns.

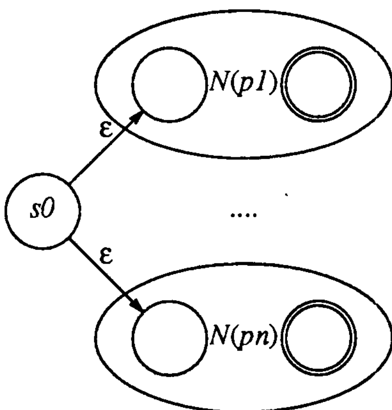


Figure 4. NFA for regular definitions p1..pn

If we simulate the above NFA and find a set of states that contains a accepting state, to find the longest match we must continue to simulate the NFA until it reaches termination. After we reach termination, we must select the longest match.

From the NFA constructed above Lex constructs a DFA pattern recognizer. It optimizes the resulting DFA, and compresses the resulting transition table. Figure 5 shows a schematic of the lexical analyzer constructed.

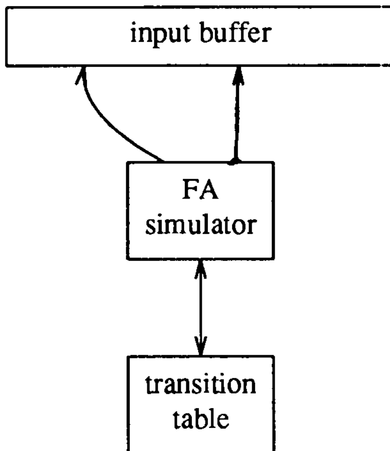


Figure 5. Schematic lexical analyzer

4. THE LEXICAL ANALYZER FOR THE NMDL++ LANGUAGE

The lexical analyzer for the NDML++ language is generated with Lex, some routines are added that support the storage of identifiers. Start conditions are used to specify the different modes the lexical analyzer can be in. If a rule is prefixed by a start condition, the pattern is only matched if the start condition is set. The lexical analyzer can be in one of four modes:

- Normal mode
- Scanning mode
- Comment mode
- include mode

4.1 The Normal mode

This is the default mode, identifiers, keywords, constants, operators, and delimiters are recognized and returned to the parser.

Identifiers consist of all upper and lower characters in the (english) alphabet, and the readability character "_". The identifiers are kept in a linked list, if a new identifier is matched it is added to the list with *new_id()*.

A keyword is a special case of identifier, an identifier is compared with all the character strings in the *keywords* array, if it matches one the keyword value found in the *keywords_val* is returned.

There are three types of constants.

- integer: Contains only digits, the minus sign is handled by the parser.
- real: All types are of reals are recognized, either exponent or scaling factor can be used.
- boolean: There are only two boolean values, TRUE and FALSE, because booleans consist out of characters they are handled as a special case of identifier.

Each operator is matched separately and has its own token, for instance "==" is the equality operator "EQOP".

There are 8 delimiters; "(", ")", ",", ".", "[", "]", ";" and ":". The token value is equal to the ACSII value of the delimiter.

A token and it's attribute are stored in a global variable *yylval*. *yylval* is of the *field_struct* type, it contains:

- The token value, each token has its own integer rank number, this can be found in defines.h.
- The token's attribute, stored in a C union. The union can contain integer, real, identifier(character pointer), keyword value (also integer), and boolean.

4.2 The scanning mode

If the -m option is not set (see [3]), the first module description is parsed. The file is scanned for system definitions that have been referenced previously, but have not been parsed. *mdl_used()* is used to determine if a module description has been referenced before. System definitions that have not been referenced before are skipped.

If a leafcell or system keyword (with mixed upper and lower case characters) is matched, the identifier following it is the system name. If the system definition is to be parsed we switch to normal mode, and start parsing the system description. Otherwise we keep on scanning the file.

4.3 The comment mode

The lexical analyzer is reading a comment, all characters are skipped. Comments can be nested, the comment level is stored in the local variable *comment_level.*

The first time a "begin of comment" denoter ("/*", "(*" and "{") is matched the lexical analyzer enters the comment mode. The current mode is saved, and *comment_level* is set to 1. Each time a new "begin of comment" denoter is matched, *comment_level* is incremented. If an "end of comment" denoter ("*/", "*)" "}") is matched, the *comment_level* is decremented, if zero the lexical analyzer switches back to the previous mode.

4.4 The include mode

If the include keyword (with mixed upper and lower case characters) is matched, the include mode is entered. The next word is retrieved and stored as an include file reference with *addfilenode()*. The lexical analyzer switches back to the scanning mode.

5. THE PARSER

The parser checks the sequence of tokens obtained from the lexical analyzer for correct syntax and generates an intermediate data-structure. If the syntax is not correct (syntax) error messages are generated that should be as selfexplanatory as possible. Because programmers frequently write incorrect programs (or simulation specifications in our case), a good compiler should assist the programmer in identifying and locating errors. Semantic errors must be handled separately, this will be discussed further on in this chapter.

5.1 The design of a language

The syntax of programming language constructs can be described by context-free grammars (grammar for short) or BNF (Backus-Naur-Form) notation. Grammars offer advantages for both language designers and compiler writers.

- A grammar has an inherently recursive structure that matches the constructs of (programming) languages. This makes it easy to understand the syntax specification of a language.
- Tools are available that check a grammar for correctness and generate a working parser for it. This ensures the correctness of a certain grammar specification.
- New constructs can be added to a language more easily when the existing implementation is automatically from an existing grammar specification, thus resulting in a shorter development time.

5.1.1 Grammars

A grammar consists of terminals, nonterminals, a start symbol, and productions.

- Terminals are the basic symbols from which strings are formed. Terminals are the tokens we discussed previously when we explained the lexical analyzer.
- Nonterminals are syntactic variables that denote a set of strings. The nonterminals define a set of strings that help define the language generated by the grammar. They impose a hierarchical structure on the language that is useful for both syntax analysis and translation.
- In the grammar one nonterminal is distinguished as the start symbol, and the set of strings it denotes is the language defined by the grammar.
- Productions of a grammar specify the manner in which the terminals and nonterminals can be combined to form strings. Each production consists of a nonterminal, followed by an arrow, followed by a string of terminals and nonterminals.

Example 5. A grammar that specifies a simple expression.

$$
\begin{aligned}
&expr \rightarrow expr\ op\ expr \\
&expr \rightarrow (\ expr\ ) \\
&expr \rightarrow \text{-}\ expr \\
&expr \rightarrow \mathbf{id} \\
&op \rightarrow +\ |\ \times
\end{aligned}
$$

The terminal symbols are **id** - + × ( ).
The nonterminals are *expr* and *op* , of which *expr* is the start symbol.
The choice between + and × for the operands is denoted by the |.

5.1.2 Derivations and parse trees

There are several ways to view the process by which a grammar defines a language. A derivation gives a precise description of the top-down construction of a legal string for a certain nonterminal.

Example 6. We use the grammar from example 5 to derive the string -(**id** + **id**) .

$$expr \Rightarrow \text{-}\ expr \Rightarrow \text{-}(\ expr\ ) \Rightarrow \text{-}(\ expr\ op\ expr\ ) \Rightarrow \text{-}(\ \mathbf{id} + \mathbf{id}\ )$$

Where ⇒ means "derives".

The operator ⇒+ means "derives in on or more steps". Given a grammar G with start symbol S, we can use the ⇒+ relation to define $L(G)$, the language generated by G. A string of terminals w is in $L(G)$ if and only if $S \Rightarrow+ w$. The string w is called a sentence.

A parse tree may be viewed as the graphical representation for a derivation, that also represents the order by which nonterminals where replaced by terminals. Each node of a parse tree is labeled by some nonterminal A, and the children of the node are labeled from left to right, by the symbols at the right side of the production A stands for. The leaves of the parse tree, when read from left to right, constitute a sentence. A parse tree constructed for the derivation of example 6 is shown in figure 6.

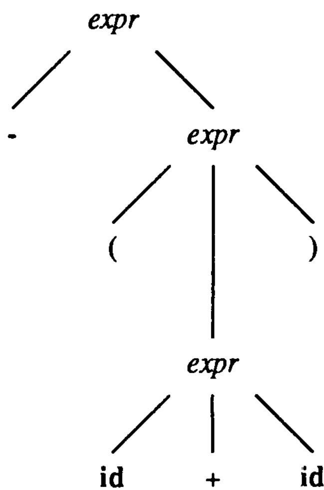


Figure 6. Parse tree for **-(id+id)**.

5.1.3 Ambiguity

A grammar that produces more than one parse tree is said to be ambiguous. It is desirable that the grammar is made unambiguous, for if not, we cannot uniquely determine which parse tree to construct for a sentence. Sometimes an ambiguous grammar can be rewritten to eliminate the ambiguity.

Example 7. Consider the following grammar.

stm → **if** *expr* **then** *stm*
  | **if** *expr* **then** *expr* **else** *stm*
  | *other*

This grammar is ambiguous because the string "**if** *E1* **then if** *E2* **then** *S1* **else** *S2* " has two parse trees. The **else** can be matched with the first, or last **if**.

In most programming languages each else must be matched with the closest previous unmatched then. We can match the last else to the previous if, if we rewrite the grammar,

stm → *matched_if_stm*
  | *unmatched_if_stm*

matched_if_stm → **if** *expr* **then** *matched_if_stm* **else** *matched_if_stm*
  | other

unmatched_stm → **if** *expr* **then** *stm*
  | **if** *expr* **then** *matched_if_stm* **else** *unmatched_if_stm*

5.1.4 Left and right recursion

Some parser generators like Yacc encourage a left recursive grammar, and others like predictive parsers encourage a right recursive grammar. Ignoring this fact can lead to larger parsers, and in some cases the parser generator will find no solution.

A grammar is left recursive if it has a nonterminal A such that there is a derivation A ⇒+ Aa for some string a. The right-recursive grammar has a nonterminal A such that there is a derivation A ⇒+ aA for some string a.

Example 8. A left recursive grammar specifying a list.

list → *item*
  | *list* **, item**

A right recursive grammar specifying the the same list would be

list → *item*
  | **item ,** *list*

It is always possible to translate a left-recursive grammar into a right-recursive and visa versa.

5.1.5 Semantics

A few syntactic construct cannot be generated by any context-free grammar.

Example 9. Consider the abstract language:

$L = \{wcw \mid w \text{ is in } (\mathbf{a}|\mathbf{b})^* \}$

All strings in L are strings of **a**'s and **b**'s separated by a **c**. It can be proven that this language is not context free. The above problem arise if a compiler is to check if a variable is declared before it is used. Because all variables are identifiers the compiler is unable to solve this problem.

A semantic checker is used instead. This program activated by the parser, compares the attributes of the declaration and instance, and reports in case of an error.

5.2 LR parsers

This section presents an efficient, bottom-up syntax analysis technique that can be used to parse a large class of context-free grammars. This technique is called LR parsing; the 'L' stands for left-to-right scanning of the input, the 'R' for constructing the rightmost derivation in reverse.

LR parsing is attractive for a number of reasons.

- LR parsers can be constructed to recognize virtually all programming language constructs for which context free grammars can be written.
- The LR parsing method can be implemented efficiently.
- A LR parser can detect a syntactic error as soon as it is possible to do so on a left to right scan of the input.

5.2.1 The LR parsing algorithm

The schematic form of a LR parser is shown in figure 7. It consists of an input string , an output, a stack, a driver program, and a parsing table that has two parts (action and goto). The driver program is the same for all parsers, only the parser tables change from parser to parser. The parser uses the stack to save the state symbols S_i, and the grammar symbols X_i.

The program driving the parser behaves as follows. The current state and grammar symbol are s_m and x_m respectively, the current input symbol a_i. The program consults action[s_m,a_i], the parsing action table entry which can have one of four values:

1. shift s: Both the current input symbol a_i and the next state s, which is goto[s_m,a_i], are pushed onto the stack.
2. reduce A → b: The parser pops as many elements from stack as there are grammar symbols on the right side of the production. The parser then pushes both A, the left side of the production, and s, the entry for goto[s_{m-r},A] onto the stack, where r is the number of grammar symbols on the right side of the production. The current input symbol is not changed. Also the the semantic action associated with the reducing production is performed.
3. accept: The parsing is complete.
4. error: The parser has discovered an error, and calls an error recovery routine.

The LR parser produces a bottom up construction of a sentence, where the current state contains all the information the parser needs.

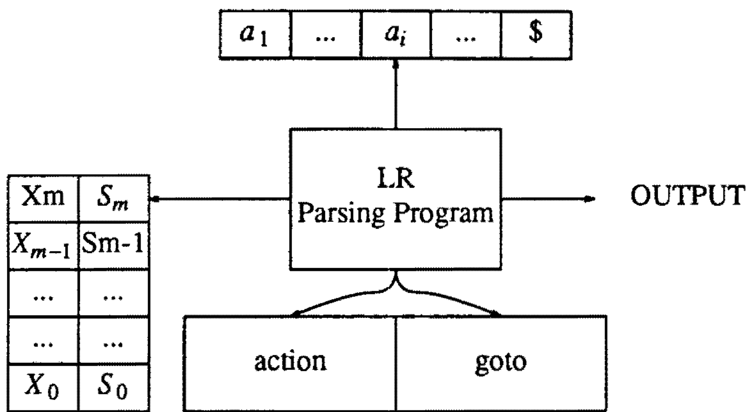


Figure 7. Model of an LR parser

5.3 Yacc, an LALR parser generator

Yacc is a LALR parser generator commonly found on UNIX, or UNIX related systems that constructs a parser from a specification in the Yacc language. The Yacc language is an extended version of the grammars discussed previously. A complete definition of the Yacc language can be found in [10],[11]. This chapter merely gives an indication how Yacc specifications can be made.

5.3.1 The Yacc specification

A Yacc source program has three parts:

- Declarations
- translation rules
- supporting C routines

There are two optional sections in the declaration part. In the first section ordinary C declarations are placed delimited by "%{" and "%}. The second part contains the declaration of the grammar tokens, and the definition of the operator precedence.

The translation rules are placed after the declarations and the first "%%". Each rule consists of an grammar production and the associated action. the action consists of a sequence of C statements. $$ and $\$_i$ are used to refer to the attribute value of the left and right ith grammar symbol.

The supporting C routines come after the second "%%". These can contain the lexical analyzer *yylex()*, and other semantic and allocation functions.

Example 10. A simple desk calculator.

```
{
#include <ctype.h>
%}
%token DIGIT
%%
line : expr '/n'                        { printf("%d0,$1);}
   ;
expr : expr '+' expr                    { $$ = $1 +$3; }
   | term
   ;
term : term '*' factor                  { $$ = $1 * $3; }
   | factor
   ;
factor : '(' expr ')'                   { $$ = $2; }
   | DIGIT
   ;
%%
yylex(){
....
}
```

The program Yacc generates for this input file will perform the operations of a simple desk calculator. It adds and multiplies DIGIT's, integer or real values. We can see how easy it is to add some new operation to the desk calculator without having to program a parser from scratch. This is a great advantage of using Yacc.

6. THE PARSER FOR THE NDML++ LANGUAGE

The parser for the ndml++ language is generated with the parser generator Yacc, extra functions are provided for semantic checks, and the construction of a parse tree. The parser Yacc generates a bottom up construction of the language, this can be used to construct a parse tree efficiently.

6.1 The parser data structures

The parser has its own data structures, the grammar symbols on the stack are the only elements that can be altered. The elementary parts of a parse tree are the nodes.

6.1.1 The grammar symbols

The grammar symbols on the parser stack are the *field_struct* structures discussed in the previous chapter. Apart from the tokens already discussed, two extra elements can be stored in a *field_struct*, a pointer to a child node, and a pointer to the first element of a list.

6.1.2 Simple parse tree nodes

A parse tree node consists of one or more *node_struct* structures linked together. Each *node_struct* consists of:

- An integer nummer representing the node type, all *node_struct*'s node type for one parse node are equal. in the .
- A *field_struct* containing a copy of a grammar symbol on the stack, this can be a link to a child node, or the first list element.
- A link to the next *node_struct* in the list.

Each *node_struct* contains a copy of the grammar symbol that make up the derivation.

Example 11: Consider two grammar rules specifying an expression:
expr → *expr* + *expr*
expr → **id** | **const**

A valid string in this language: '10' '+' 'i' '+' '30'.
The right most construction of a parse tree for this string is shown in figure 8. The big boxes are the *node_struct* structures, and the contents of the *field_struct* structures is printed in the smaller boxes.

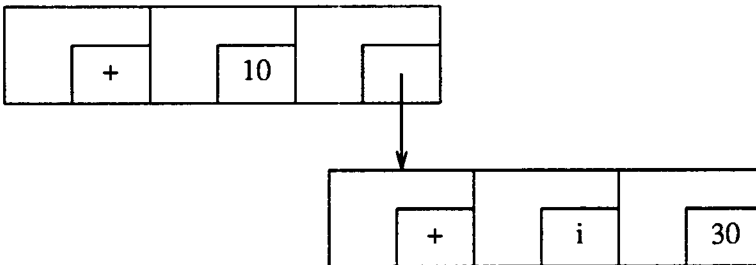


Figure 8. Parse tree for the string: '10' '+' 'i' '+' '30'

6.1.3 List nodes

The grammar describing the NDML++ language contains a large number of lists. A list consists of one or more grammar symbols, terminated or separated by a delimiter or separator respectively. Normally we would create a new node for each element in the list, but this would lead to a large number of nodes. Instead a new node type is introduced, the "list node". A list node contains two *node_struct* structures, the first contains a pointer to the first element in the list, and the second contains the delimiter or separator. The elements in the list are nodes with only one *node_struct* , it contains a single token or a link to a child node. In figure 9 an example of a list node for the string "'a' ',' 'b'" is shown.

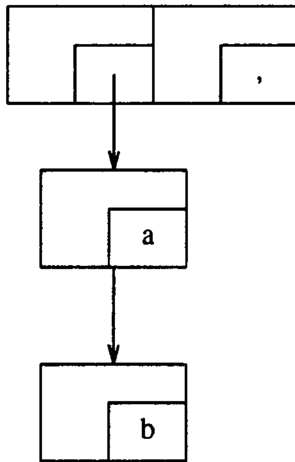


Figure 9. Parse tree for a list node

6.2 Construction of the parse tree

The syntax diagrams or EBNF (Extended Backus Naur Form) describing NDML++ [2], can easily be translated to a yacc grammar. The NDML++ language is designed to be LL(1), yacc will have no problems in generating a compiler. Lists are constructed right recursive, this will prevent the parser stack from growing to large. Operator precedence parsing is used to define the operator precedence level, and left or right associativity.

The parser constructs the language bottom-up, the leaves are reduced first, then the rest of the tree is constructed. This means that the grammar symbols of a reduction will only contain terminals, and nonterminals for which a parse tree has already been constructed.

6.2.1 A simple node

When a simple grammar rule reduced a new node is created by *make_node()*.

For each grammar symbol at the right of the -> a *node_struct* is allocated, and the contents of the *field_struct* are copied. Empty terminals, used to parse optionals, are skipped. The *node_struct*'s are linked together, and the node type of each is set to the node type that is passed to *make_node()*. A *field_struct* is allocated, and is made to point to the first *node_struct* of the newly created node. This *field_struct* is returned by *make_node()*, and the parser puts it on stack as the current grammar symbol.

6.2.2 A list node

When the first element of a list is reduced *make_list_node()* is used to create a new list node. New elements are added to this list with *add_to_list_node()*.

make_list_node() allocates two nodes. The first node contains only one *node_struct*, a copy of the token on the stack is put in the *field_struct*. The second node consists out of two *node_struct*'s, the first contains a pointer to the first node in the list, and the second contains the separator.

add_file_node() scans the previously created list, and adds a new node containing a copy of the token on the stack ,at the end of the list.

6.3 Error recovery

Yacc provides a method that helps the compiler writer to construct an efficient error recovery method. When the parser detects an erroneous lookahead token for the current state, it calls the error routine *yyerror()*, furthermore it puts the "error" token on the parser stack. This token can be used in grammar rules, it specifies where the error recovery should find place. Normally the parser remains in the error state until three tokens have been parsed successfully. If the compiler writer is able to design its grammar (containing the error symbol) that provides a quicker error recovery, the function *yyerrok* can be used to reset the error state.

We place the error symbols as close as possible to the terminals of the grammar. A method found in [13] is used to recover errors in lists, with no loss of information.

Example 12: Error recovery used for list.

Consider a list of y's separated by T.

```
x : y
 | x T y      {yyerrok;}
 | error
 | x error
 | x error y  {yyerrok;}
 | x T error
 ;
```

Between each grammar symbols at the right side of a production, a statement is placed that places the expected symbol in a string. This string is printed when the parser finds an error, and calls the error routine *yyerror()*. Yacc provides no means to do this any other way.

7. THE TRANSLATOR

When all module descriptions have been parsed successfully, the parse trees constructed by the parser are being translated into one simple NDML description[12]. The translation process is fairly simple.

- The root module is translated first, all instances made by the root module are kept in a linked list.
- After the root module is completely translated, the instances in the linked list are processed.
- These instances can also induce new instances, these are added to the linked list.
- The above process continues until the linked list is empty, and all instances have been translated.

We use the top-down form of the parse tree for two recursive functions. *eval_node* translates a parse tree node, and *eval_field* translates it's fields. Both functions contain references to each other, which induce the recursive process.

7.1 The data structures

The module instances are kept in a linked list which the global *top_of_mdl_stack* points to.
Each module instance consists of:

- A link to the module description (parse tree).
- A parameter set number, all different parameter sets of a certain module description are numbered starting from "1".
- A link to the next module instance in the list.

When a module is instanciated with a nonexisting parameter set, a parameter_set_info structure is generated. This structure is linked to the prm_sets of the module description. This list is used to check if a certain parameter set already exists.
The parameter_set_info contains:

- The parameter set number, this value corresponds with the parameter set number of the module_instance.
- A pointer to a parse tree that represents the actual parameter assignment.
- A link to the next parameter_set_info in the list.

7.2 translation of the parse tree fields

Parse tree fields are translated by *eval_field()*. Each field type has its own associated action:

- Real constant: real: the real value is printed, the accuracy can optionally be specified by the user using the -f option.
- Integer constant: print integer value.
- Boolean constant: print "FALSE" or "REAL".
- Identifier: print the character string.
- Keyword: compare the keyword value with the elements of the *keyword_val* array, print the character string in the corresponding place of the *keywords* array.
- Separator: print the character corresponding to the ASCII value.
- System name: print the system name followed by a "__" and the current parameter ste number.
- Parameter: get parameter value, the parameter can be an actual parameter, a default value, and a local parameter (FOR NEXT statement).
- Empty: nothing is printed.
- Subnode: *eval_field()* is called for the node the subnode points to.

If in any of this cases the contents of the field are incompatible with their field type, an error message is issued.

7.3 Translation of the most common node types

For a small number of parse tree nodes the result of the translation process for a specific node (not its childs) is identical to the initial specification.

For the simple node type we evaluate all fields of the node in succession.

For a list each item of the list is printed, separated or terminated by the separator respectively the terminator.

7.4 Functions and expressions

The result of an expressions or function must be calculated before it can be printed, the result is calculated by the C function *calc_expr()*. The arguments of an expression or function can be any type of constant or parameter value or a sub-expression, the arguments are fetched by *get_operand()*. The function or expression type determines which types are valid, for instance booleans may not be used in arithmetic functions like LOG() and EXP(). There are two types of expressions, dual expressions with two arguments, and unary with just one.
Dual expressions are divided into three groups:

- Multiplicative operations: the operands must be integer or real, the result is integer if both operands are integer, and real otherwise. The valid operations are multiplication (×), division (/), addition (+), subtraction (-), and power (××).
- Comparative operations: the operands can be integer or real, the result is boolean. The valid operations are equal (==), not equal (!=), less or equal (<=), greater or equal (=>), less (<), and greater (>).
- Binary operations: both operands and result are boolean. The valid operations are or (OR), and and (AND).

There are only two unary expression types. The negation (-) calculates the negated value of an integer or real, and the not (NOT) operation calculates the logical negation of a boolean value.

The result of a function is calculated by *calc_function*. The operand can be integer or real, the result has the corresponding type.
The following functions have been implemented:
sine (SIN), cosine (COS), tanges (TAN), arcsine (ARCSIN), arccosine (ARCCOS), square root (SQRT), exponent (EXP), natural logarithm (LN), ^{2}log (LOG2), and the ^{10}log (LOG).

A list function processes a list of arguments, arguments must be integer or real. Currently two list function are implemented, maximum of list (MAX), and minimum of list (MIN).

7.5 Indices and dimension specifications

One of the main tasks of the compiler is to translate vectorized terminals, and instances into lists of single elements. *make_node* processes such elements separately. The internal data structure *VECTOR* is used to store vector dimensions. The *VECTOR* contains the dimension of the vector, and an array with the size of each dimension.

Declarations of terminals, instances, and nets with dimension specifications are evaluated by *eval_decl_with_dim_spec*. A list of all the vector items is printed, separated by commas. The first index is always [0,..,0], and the vector specifies the lenght. This is the same method used in the C programming language, only dimension specifications have an arbitrary number of indices. For instance, net[2,3] becomes, "net[0,0],net[0,1],net[0,2],net[1,0],net[1,1],net[1,2]".

System instances and formal terminals with an index are evaluated by *eval_sys_inst_with_index*, and *eval_fml_term_with_index*. The index is checked against the dimension specification of the formal terminal or instance, and if correct is printed.

7.6 System instances and terminal interconnect

System instances and terminal interconnect statements operate on a list of actual nets.
There are four types of actual nets.

- Empty net "-": the terminal is not connected, the terminal size is unspecified.
- A simple actual net: just one net or terminal, the terminal has dimension 0.
- Actual net subrange: a subrange of a vectorized actual net is specified.
- Bundle: a bundle of actual nets.

7.6.1 Actual net data structures

Two internal data structures *ACT_NET* and *ACT_NET_ITEM* are used to evaluate actual net specifications.

An *ACT_NET_ITEM* contains an actual terminal name and an optional extension that contains a link to a vector specifying the indices, and a link to the next *ACT_NET_ITEM*.

An *ACT_NET* contains a link to the actual net items, a link to a vector specifying the size of the actual terminal, and a link to the next *ACT_NET*.

7.6.2 evaluation of actual subranges

The validity of the subrange is first checked against the corresponding net or terminal declaration. The actual subrange is then translated into a *ACT_NET* containing the actual net size, and a link to a list of *ACT_NET_ITEMs*. An *ACT_NET_ITEM* is created for each element in the subrange, and are linked into a list.

7.6.3 evaluation of bundles

The bundle concept is used to create a actual net with correct dimensions from a list of actual nets. The resulting actual net can be connected to a vectorized terminal. The NMDL++ manual [1] describes the different bundle types and their meanings. Four types of bundles can be constructed, horizontal stack, vertical stacks, horizontal catenate , and vertical catenate. The C function *get_bundle()* translates a list of actual nets into the resulting actual net, applying the specific bundling rules. Each bundle is constructed in the same way, first the size is calculated by combining the size of the actual terminals, then the *ACT_NET_ITEMs* of the actual nets are linked into a new list following a certain algorithm.

7.6.4 System instance declarations and system instance invocations

The system declaration specifies the system template and parameters that are to be used for a list of system instances.

eval_inst_decl() prints the system instances by calling *eval_field()* with a pointer to the instance list. The parameter set is replaced by the version number of the parameter set. The template name followed by "__", and the parameter version number is printed.

A system instance connects the actual nets with the terminals of the instanciated module. Both actual nets and terminals can be vectorized, their sizes must be the equal. A system instance declaration is processed by *eval_inst_decl()*.

The template name and the parameter set number are found with the corresponding system instance declaration. These are used to get the module description (parse tree), and actual parameter set. The actual nets are linked into a list by *get_act_net_list()*. Current parameters and parameter template are replaced by the parameter set, and parameter template of the instance.

The dimensions of the actual nets and terminals are compared, and the actual nets are printed. The parameter template and current parameters are restored to their initial value.

7.6.5 The connect statement

The connect statement connects a number of actual terminals, optionally the actual terminals are vectorized. If so the connect statement must be broken down to simple connect statements the simple NDML parser can read. The connect statement is processed by *eval_connect_stm()*.

If there are any bundles or subranges these are evaluated by there respective functions. The sizes of all actual terminals must be equal, the empty net complies its size to the other actual nets. Then a connect statement is generated for each *ACT_NET_ITEM* in the list.

7.7 Conditional and repetitive statements

Currently two conditional statements are implemented, an if then else and a case like construction.

The if then else statement is evaluated by *eval_if_then_else*, it evaluates the condition with *eval_logical_expr()*, and evaluates either the first or the last statement list.

The case like construction is evaluated by *eval_select_stm()*, the clause for which the condition evaluates to "TRUE" is evaluated, if none of the conditions evaluates to "TRUE" the otherwise part (default) is evaluated.

A logical extension that comes with the possibility of vectorizing terminals and nets, is the for_next statement. A local parameter is created to hold the current value, this value can be obtained by *get_prm()* when it calls *get_local_prm_val()*. Local parameters are kept in a linked list, it is possible to create loops inside loops etc. The C function *eval_for_next_stm* creates a new local parameter value, and initializes it. The system statement list is evaluated, and the local parameter value is incremented. This process continues until the stop condition evaluates to "TRUE", the local parameter value is removed and the function returns.

REFERENCES

1. Janssen G.L.J.M. ; "Network Description & Modeling Language - NDML" ; The Integrated Circuit Design book, Delft University Press, pp.4.60-4.108, 1986.
2. van Gemert Q.J.A. ; "Extended Network Description & Modeling Language (NDML++)";
3. van Eindhoven J.T.J. ; "The Piecewise Lineair Simulator"; The Integrated Circuit Design book, Delft University Press, pp.4.1-4.59, 1986.
4. de Graaf A.C; Janssen G.L.J.M.; "Proposal for a Network Description Format in the ICD-System"; The Integrated Circuit Design book, Delft University Press, pp.1.49-1.60, 1986.
5. Kernighan B.W. ; Ritchie D.M. ; "The C programming Language" ; 1st ed. Englewood Cliffs, NJ, Prentice Hall, 1988.
6. Johnson S.C. ; "Lint, a C Program Checker" ; Comp. Sci. Tech. Rep. No. 65, Dec 1977.
7. "Lint, a C Program Checker"; HP-UX Concepts and Tutorials, Vol 1.
8. Lesk M.E. ; "Lex, A Lexical Analyzer Generator" ; Comp. Sci. Tech. Rep. No. 39, AT&T Bell Labs., Murray Hill, New Jersey, Oct 1975.
9. "Lex, A Lexical Analyzer Generator" ; HP-UX Concepts and Tutorials, Vol 1.
10. Johnson S.C. ; "Yacc, Yet Another Compiler-Compiler" ; Comp. Sci. Tech. Rep. No. 32, AT&T Bell Labs., Murray Hill, New Jersey, 1975.
11. "Yacc, Yet Another Compiler-Compiler" ; HP-UX Concepts and Tutorials, Vol 1.
12. Hopcraft J.E. ; Ullmann J.D. ; "Formal Languages and their relation to automata"; Addison-Wesley, Reading, Mass, pp.22-23, 1969
13. Schreiner A.T. ; Friedman H.G. ; "Introduction to Compiler Construction with UNIX"; Prentice Hall Inc, Englewood Cliffs, NJ 07632, pp.65-81, 1985.

The **NDML** Network Description and Modeling Language

Q.J.A. van Gemert

J.T.J. van Eijndhoven

Design Automation Section (ES)

Eindhoven University of Technology

The Netherlands

CONTENTS

1. INTRODUCTION 2
1.1 Design Goals 2
1.2 Language Summary 2
1.3 Notation 3
1.4 Classification of Errors 4

2. LEXICAL ELEMENTS 5
2.1 The Source Character Set 5
2.2 Tokens and Spacing Conventions 6
2.3 Comments 8
2.4 Extension Mechanism 9

3. NDML++ DESCRIPTION HIERARCHY 10
3.1 Include Files 10
3.2 Libraries 10

4. EXPRESSIONS AND FUNCTIONS 11
4.1 Binary Expressions 11
4.2 Unary Expressions 12
4.3 Operator Precedence And Associativity 12
4.4 Functions 13

5. SYSTEM INTERFACES 14
5.1 Terminal Declarations 14
5.2 Formal Parameters 16

6. LEAFCELL SYSTEMS 18
6.1 The Select Statement 18
6.2 The Matrix Definition 19
6.3 The Voltage Reference 19
6.4 The Variable Definitions 20
6.5 The Matrix Rows 21
6.6 Assignment Statement 22
6.7 Remove Statement 22
6.8 Pivot Statement 22
6.9 Optional Leafcell Statement 22
6.10 Iterative Leafcell Statement 23

7. COMPOUND SYSTEMS 24
7.1 Instance Declarations 24
7.2 Net Declarations 26
7.3 The Compound Body 27
7.4 Instance Invocation 28
7.5 Net Connection 29
7.6 Optional Compound Statement 30
7.7 Iterative Compound Statement 30

8. NDML language evolution 31
8.1 Recent NDML extensions 31
8.2 Incompatibilities with previous NDML 31
8.3 Plans for future extensions 32

9. REFERENCES 33

APPENDIX 1: NDML++ EXAMPLES 34

1.1 A CMOS ring oscillator 34
1.2 A leafcell max 34
1.3 A leafcell capacitor 35
1.4 An AD converter circuit 36

APPENDIX 2: NDML++ SYNTAX DIAGRAMS 37

1. INTRODUCTION

This report describes the extended Network Description and Modeling Language NDML++. NDML++ contains some improvements over the original language NDML. This language designed by G.L.J.M. Janssen is described in [Janssen].

The network description part of the language provides general constructs for expressing hierarchical network structures, i.e. system types called compounds may be defined by interconnecting a number of more simpler circuits.

The modeling part of the language allows the circuit primitives to be described by means of a set of piecewise linear equations and ordinary first order differential equations. Circuit primitives thus modeled are called leafcells.

Both descriptions, compounds and leafcells, share a common interface construct. In this way the implementation information of a circuit template is hidden. The language provides parameter mechanisms for modifying the circuit behaviour.

The range of circuits that can be modeled solely depends on the descriptive power of the leafcells (structural description can not introduce new behaviour).

1.1 Design Goals

The primary goal for NDML is to provide the Eindhoven Piecewise linear simulator PLATO with a textual user-interface for describing network input data. This opposed to ESCHER which has a graphic user-interface for the network interface, see [ESCHER]. The language should both be easy to learn and easy to implement. This is achieved by carefully designing the syntax, including suggestive reserved words and using similar constructs to indicate similar semantics.

The step-by-step design process used for NDML++ is described in [Ruehli]. Similar ideas on a structure description language can be found in [BDL].

1.2 Language Summary

A NDML circuit description is composed of one or more source text files that are linked by include file references. Each file consists of one or more system definitions. System definitions can either be compound definitions or leafcell definitions.

A leafcell definition describes the behaviour of a piece of hardware by means of a piecewise linear model. Leafcell definitions are normally not supplied by the user, the models should be prepared by qualified person(s) and collected in libraries.

A compound definition specifies a template for a hardware module by specifying the interconnection between the sub-systems it is built of. Compound systems can then be used as sub-system in other compound definitions. The use of a system inside another is called instantiation. A sub-system is said to be an instance derived from a template. Commonly used compound definitions are also collected in libraries.

A system definition starts with an interface specification. Here a name for the template is introduced and its terminal ports are named and typed. Optionally, parameters can be associated with this definition.

Terminal ports represent the physical contacts of a hardware component. Often they are called pins.

Parameters are convenient to model certain varying quantities of the system. Boolean parameters are used to trigger optional constructs.

To describe the contents of a compound system, instance declarations are required that state the number and kind of sub-systems. At this point parameter values for the instances must be given, either explicitly by means of assigning a value obtained by evaluating an expression, or implicitly by the default mechanism.

The connections between the pins of the instance (sub-) systems is achieved by means of nets. A net is said to connect a number of terminal ports. Nets are denoted by names and declared just prior to the connection statements. Terminal ports can can also be used as an implicit form of net.

1.3 Notation

This manual presents the language concepts in a bottom-up fashion, i.e. first the basic character set (alphabet) is defined, then the lexical rules that guide the formation of elementary language symbols (words) are discussed and finally the syntax and semantics is presented per basic concept. To define the syntax an extended Backus Naur Form is used. The EBNF we use closely resembles the proposal in [Wirth] also known as Wirth's Syntax Notation. However our extensions are specially made to allow for automatic processing, e.g. to determine the type of grammar and to generate syntax diagrams. The Extended BNF meta-language we use to describe the syntax can be defined in itself:

```
<syntax>          ::= { <production-rule> } .

<production-rule> ::= <nonterminal-symbol> "::=" <expression> "." .

<expression>      ::= <term> { "|" <term> } .

<term>            ::= { <factor> }+ .

<factor>          ::= <nonterminal-symbol>
                    | <terminal-symbol>
                    | "(" <expression> ")"
                    | "{" <expression> "}" [ "+" ]
                    | "[" <expression> "]" .
```

The meta-symbols have the following meaning:

<table>
<tr><th>Meta-symbol</th><th>Meaning</th></tr>
<tr><td>::=</td><td>devides a production rule in a grammar symbol, and its definition.</td></tr>
<tr><td>.</td><td>denotes the end of a production rule.</td></tr>
<tr><td>T1 | T2</td><td>denotes a choice between either T1 or T2.</td></tr>
<tr><td>[E]</td><td>denotes an optional construct E.</td></tr>
<tr><td>{ E }</td><td>denotes repetition of E zero or more times.</td></tr>
<tr><td>{ E }+</td><td>is equivalent to E { E } and thus denotes a repetition of one or more times.</td></tr>
<tr><td>(E)</td><td>parenthesis can be used to override the operator precedence, concatenation of factors is denoted by juxtaposition.</td></tr>
</table>

To unambiguously denote a literal which usually is called a terminal symbol, it is written between double-quote characters. For clarity the syntactic variables or non-terminal symbols stand out by enclosing them in angular brackets ("<" and ">").

1.4 Classification of Errors

The language recognizes three categories of errors.

1. Errors that must be detected at compilation time. These errors correspond to any violation of a rule of the language, other than stated below. All lexical and syntactic errors belong to this class.
2. Errors that must be detected at evaluation time. These errors result from violations against language rules that are outside the scope of a compiler, e.g. errors during expression evaluation, cross-checking separately compiled modules etc.
3. Errors that result from violations against the pragmatics of the language. The language allows for certain circuit descriptions that have no physical counterpart, like a circuit with only one electrical contact. Also violations against certain connection rules belong to this class. These kinds of errors need not necessarily be detected by a compiler. If a erroneous description is evaluated and/or simulated, its effect is undefined and unpredictable unless stated otherwise.

2. LEXICAL ELEMENTS

This chapter defines the lexical elements (tokens) of the language. Rules will be presented that determine the creation of tokens from individual characters. Furthermore the use of text formating characters like space, TAB, and new-line will be explained. Comments and the extension mechanism are not part of the language, and are added for convenience of the user only.

2.1 The Source Character Set

All language text may be represented with a basic character set, which is subdivided as follows:

(a) upper case letters

```
<uppercase_letter> ::=
      "A"|"B"|"C"|"D"|"E"|"F"|"G"|"H"|"I"|"J"|"K"|"L"|"M"
    | "N"|"O"|"P"|"Q"|"R"|"S"|"T"|"U"|"V"|"W"|"X"|"Y"|"Z" .
```

(b) digits

```
<digit> ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9" .
```

(c) special characters

```
<special_character> ::=
      "!" | "&" | "(" | ")" | "*" | "+" | "," | "-"
    | "." | "/" | ":" | ";" | "<" | "=" | ">" | "["
    | "]" | "_" | "{" | "|" | "}" | "%" .
```

(d) the space, tab, and new-line characters

The character set may be extended to include further ASCII characters. These are:

(e) lower case letters

```
<lowercase_letter> ::=
      "a"|"b"|"c"|"d"|"e"|"f"|"g"|"h"|"i"|"j"|"k"|"l"|"m"
    | "n"|"o"|"p"|"q"|"r"|"s"|"t"|"u"|"v"|"w"|"x"|"y"|"z" .
```

(f) other special characters

```
<other_special_character> ::=
      """" | "#" | "$" | "'" | "~"
    | "?" | "@" | "\" | "^" | "`" .
```

A lower case letter is distinct from the corresponding upper case letter (unless the -u option is set). However it is considered bad style if two names only differ in the case of their letters. Other special characters can only appear in comments and the extension construct.

2.2 Tokens and Spacing Conventions

A NDML++ description is a sequence of tokens, the partitioning of the sequence into lines and the spacing between tokens does not affect this in any way. The tokens are names, reserved words, numeric literals, operators and separators.

Adjacent lexical elements may be separated by spaces or by the passage to a new line. A name or numeric literal must be separated in this way from an adjacent name or numeric literal. A space must not occur within a token, each token must fit on one line.

Certain ASCII control characters may be used to achieve a proper text layout. Passage to a new line is defined by the implementation. On UNIX based systems the C-language newline is preferred. Horizontal tabs are allowed and when they occur between lexical units they are equivalent to a space. Any other control character is flagged as an illegal character by an appropriate error message.

2.2.1 Operators and Separators

Operators are mainly used in expressions, they indicate the operation that should be performed on the operand(s).

The operators are

<simple_operator> ::= "!" | "" | "+" | "-" | "/" | "<" | "=" | ">" | "%" .*

or one of the following compound operators

*<compound_operator> ::= "**" | "<>" | "<=" | ">=" | "&&" | "||" | ":=" | "==" | ".."*

Separators are used to group syntactic entities, sometimes there only function is to improve readability. The separators are

<separator> ::= "(" | ")" | "[" | "]" | "," | ";" | ":"

or one of the following compound separators

<compound_separator> ::= "<<" | ">>" | "<_" | "_>" | "</" | "/>"

2.2.2 Names

Names are used to identify objects, two names refer to the same object when they are spelled identically, including the case of all characters. A name is a sequence of letters, digits, and underscores. A name must begin with a letter and may not have the same spelling as a reserved word. All characters are significant (even underscores), however if the -u option is set upper- and lower-case characters are considered the same.

<name> ::= <letter> { <letter> | <digit> | <readability_character> }.

<readability_character> ::= "_" .

<letter> ::= <lowercase_letter> | <uppercase_letter> .

Some implementations may restrict the number of significant characters in an name, the NDML++ compiler has a 100 character limit.

Examples of names:

x bool_and2 NDML

2.2.3 Reserved Words

The names listed below are called reserved words and are reserved for special significance in the language. Declared names may not be reserved words.

ARCCOS	ARCSIN	ARCTAN	BEGIN	CASE
CONNECT	COS	DEFAULT	DU	ELSE
EXP	END	FALSE	FI	FOR
I	IF	INITIAL	INSTANCE	LN
LOG	LOG2	LOG10	MAX	MIN
NET	NEXT	OTHERWISE	PIVOT	PL
REFERENCE	REMOVE	ROUND	SIN	SQRT
SYSTEM	TAN	THEN	TRUE	U
UNTIL	V	VAR	ZERO	

of these the following can only appear in compound system definitions:

CONNECT INSTANCE NET

whereas,

CASE	DU	ELSE	I
PL	PIVOT	OTHERWISE	REMOVE
VAR	U		

can only appear in leafcell system definitions.

Upper and lower case spellings of a reserved word are not distinct.

2.2.4 Numeric Literals

There are three classes of numeric literals: integer, real and boolean.

<constant> ::= <numeric_literal> / <boolean_literal> .

<boolean_literal> ::= "FALSE" / "TRUE" .

<numeric_literal> ::= <unsigned_integer_number_literal>
      / <unsigned_real_number_literal>

<unsigned_integer_number_literal> ::= <digit_sequence> .

<digit_sequence> ::= <digit> { <digit> } .

<unsigned_real_number_literal> ::=
  <digit_sequence> <scientific_scale_factor> [<digit_sequence>]
 / <digit_sequence> <fractional_part> [<scale_factor>] [<scale_factor>]
 / <digit-sequence> <scale_factor> [<scientific_scale_factor>] .

<fractional_part> ::= "." <digit_sequence> .

<scale_factor> ::= ("E" / "e") ["+" / "-"] <digit_sequence> .

<scientific_scale_factor> ::=
<Tera> | <Giga> | <Mega> | <Kilo> | <milli>
| <micro> | <nano> | <pico> | <femto> .

```
<Tera>    ::= "T" .   (* 1E+12 *)
<Giga>    ::= "G" .   (* 1E+9  *)
<Mega>    ::= "M" .   (* 1E+6  *)
<kilo>    ::= "K" .   (* 1E+3  *)
<milli>   ::= "m" .   (* 1E-3  *)
<micro>   ::= "u" .   (* 1E-6  *)
<nano>    ::= "n" .   (* 1E-9  *)
<pico>    ::= "p" .   (* 1E-12 *)
<femto>   ::= "f" .   (* 1E-15 *)
```

Examples of numeric literals:

```
TRUE     FALSE                  (* boolean *)
0        123         32767      (* integer *)
0.0      0.0123      12345.0    (* real *)
123E-1   0.0123e12   0.3E+6     (* real with exponent *)
123M     1K1         1.23u      (* real with scale factor *)
```

2.3 Comments

Comments may appear before and after any token. They have the same delimiting effect as a space. Comments have no effect on the meaning of network and modeling description. Two types of comments are allowed "(*" and ")*", and C-like comments ("/*" and "*/"). Both types of comments may be nested!!

```
<comment> ::= "/*" <comment_string> "*/"
        | "(*" <comment_string> "*)" .

<comment_string> ::= { <a_printable_ASCII_character> }
    | { <a_printable_ASCII_character> } <comment_string> { <a_printable_ASCII_character> } .

<a_printable_ASCII_character> ::=
        <letter> | <digit> | "TAB" | "EOL" | " "
      | <special_character>
      | <other_special_character> .
```

Examples of comments:

```
(* This is a comment *)

/* Comments may extend over
   the end of a line
*/

(* This is an example of a (* nested comment *) *)
```

2.4 Extension Mechanism

The extension construct provides a well-defined escape possibility out of the language, i.e. any text within curly braces is not considered part of the language and will typically be treated just like comment. It allows different language processors to share a common program description where some processor may define and use the text in the extension construct.

<extension_construct> ::= "{" { <a_printable_ASCII_character> } "}" .

The syntax and semantics of the contents of an extension is not defined.

3. NDML++ DESCRIPTION HIERARCHY

Like in most programming languages NDML++ descriptions can be distributed over multiple text files. The links between the separate text files express the description hierarchy . The description hierarchy is expected to be top-down, first the main system description is described, then its descendants etc. This means that no system description is defined before it is referenced (except the main system description of course). Recursive definitions are not allowed, for a hardware simulation description can not contain itself.

Description module names are global, even when the descriptions are located in different files they must be distinct.

Links between text files can be implicit or explicit. Explicit links are created with include statements, implicit links are created by the compiler when the libraries are checked for missing NDML++ descriptions.

3.1 Include Files

Explicit links between the different NDML++ text files are created with include statements.

<include_statement> ::= "INCLUDE" <file_name> .

The file name can be any valid file name on the current system. Multiple include statements can be put on one line, but each file name must be preceded by the "include" keyword.

An include statement states that the module descriptions contained in the referenced text file can be used in the system descriptions defined after the include statement. The include file is not copied to the input of the compiler (as with most C-compilers) but a link is created, and whatever information is needed is extracted from the file. Include files itself can also include other files, but the information in these files cannot be used by the parent of the include file directly.

3.2 Libraries

Also library modules designed by experienced users can be used. Module descriptions that are absent from a text file and its include file(s) can be found in the libraries.

Currently three library module types can be used.

NDML++ modules: These files are accessed through the environment variable PWL_PATH (containing a list of UNIX path names). The file names are constructed by appending the module name, and "/ndml" to the path name(s). Instead of "/ndml/", "pwl_model" can be used also in order to ensure compatibility with earlier releases. These text files contain a module description in NDML++.

ICD modules: These files are also accessed through the environment variable PWL_PATH. The file names are constructed by appending the module name, and "/call" to the path name(s). These files contain a module description according the ICD standard [Graaf]. The *icd2ndml* program is used to convert these files to NDML++ format.

Delft Database These files are accessed through the environment variable PWL_DBPATH (containing a list of project names). The module description is extracted by the *xndml* program from the first project which defines the missing module.

Library modules can contain unsolved references that can be found in other library modules. Include statements can be used also, however this is not recommended.

4. EXPRESSIONS AND FUNCTIONS

An expression is a formula that specifies the computation of a value.

<expression> ::= <binary_expression>
/ <unary_expression>
/ "formal_parameter_name"
/ <constants>
/ <function> .

All types of constants can be used : real, boolean, and integer.

The only non-constant primitive operands are the formal parameters. The formal parameter value is either a actual parameter value or the default value if no actual parameter is specified.

4.1 Binary Expressions

A binary expression is constructed by applying an operator to two operands.

<binary_expression> ::= <expression> <add_op> <expression>
/ <expression> <rel_op> <expression>
/ <expression> <mult_op> <expression>
/ <expression> <pow_op> <expression>
/ <expression> <logic_op> <expression>.

The type of the operands determine the allowable operators. The combination of the operands and the operator determine the type of the result.

4.1.1 Additive and Multiplicative Operators

Both multiplicative and additive operators are left recursive. Multiplicative operators have a higher precedence than additive ones.

<add_op> ::= "+" / "-" .

<mult_op> ::= "/" / "" / "%".*

The binary operator "+" indicates addition. The binary operator "-" indicates subtraction. Both additive operators are commutative and associative. The binary operator "/" indicates division. When division by zero occurs the compiler will generate an error message. The binary operator "*" indicates multiplication. The binary operator "%" indicates the remainder, both operands must be integer. All multiplicative operators are commutative and associative.

Only reals and integers can be used as operands. If both operands are integer the result will be integer too, otherwise the result will be real. Standard conversion will be performed to fit a floating point result into an integer.

If the result of the addition, subtraction, or multiplication of two integers can not be represented by an integer, the result will be converted to real and a warning message is printed. If the result of an operation on two reals produces a value that cannot be represented by a real value, a warning message will be generated.

4.1.2 Relational Operators

The binary relational operators all have the same precedence and are left-associative.

<rel_op> ::= "<" | ">" | "<=" | ">=" | "<>" | "==" .

The relational operands check the operands for a certain relationship. If the relationship is satisfied the result will be "TRUE" otherwise it will be "FALSE". The operator "<" tests for the relationship "is less than", ">" tests "is greater than", "<=" tests "is less than or equal to", ">=" tests "is greater than or equal to", "<>" tests "is not equal to", and "==" tests "is equal to".

Only reals and integers can be used as operands for all relational operands, use of boolean values is only allowed with the "==" and "<>" operators.

4.1.3 Logical Operators

Both logical operators have the same precedence and are left-associative.

<logic_op> ::= "&&" | "||" .

The binary operator "&&" indicates the logical (boolean) and operation. The binary operator "||" indicates the logical (boolean) or operation.

Both operands must be boolean, the result is also boolean.

4.1.4 Power operation

*<pow_op> ::= "**" .*

The power operand computes the first operand raised to the power of the second operand. Integers and reals can be used as arguments, only when both operands are integer the result is integer.

4.2 Unary Expressions

<unary_expression> ::= ("!" | "-") <expression> .

The "!" stands for the logical not operation, it inverts a boolean value. Integer and real values cannot be used as operand.

On the other hand the unary minus computes the arithmetic negation of an integer or real operand. The result of this operation will inherit the operands type.

4.3 Operator Precedence And Associativity

The operators are listed from highest to lowest priority, operators with the same priority are listed on the same line.

<table>
<tr><th>Operator(s)</th><th>associativity</th></tr>
<tr><td>"**"</td><td>left</td></tr>
<tr><td>"!"</td><td>left</td></tr>
<tr><td>* / %</td><td>left</td></tr>
<tr><td>+ -</td><td>left</td></tr>
<tr><td>< > == <> <= >=</td><td>left</td></tr>
<tr><td>&& ||</td><td>left</td></tr>
</table>

4.4 Functions

A function is a special case of expression. Most functions operate on a single operand, two special functions operate on a list of operands.

<function> ::= "SIN" "(" <expression> ")"
/ "COS" "(" <expression> ")"
/ "TAN" "(" <expression> ")"
/ "ARCCOS" "(" <expression> ")"
/ "ARCSIN" "(" <expression> ")"
/ "ARCTAN" "(" <expression> ")"
/ "ROUND" "(" <expression> ")"
/ "LN" "(" <expression> ")"
/ "LOG2" "(" <expression> ")"
/ "LOG10" "(" <expression> ")"
/ "SQRT" "(" <expression> ")"
/ "EXP" "(" <expression> ")"
/ ("MIN"/"MAX") "(" <expression_list> ")" .

<expression_list> ::= <expression> { "," <expression> } .

SIN,COS,TAN,ARCSIN,ARCCOS, and ARCTAN are trigonometric functions whose names speak for themselves. LOG10 computes the base-10 logarithm, LN the natural logarithm (base-e), and LOG2 the base-2 logarithm. EXP computes the exponential function, SQRT the square root. ROUND calculates the nearest integer value for a floating point value.

All functions operate on integers and reals, the result will always be real, except for the ROUND function which always returns an integer. Any overflow, underflow, and illegal operation of functions will be reported by the compiler.

The MAX and MIN functions compute the maximum respectively the minimum of a list of operands. Operands can be both real or integer, only when all operands are integer the result will be integer.

5. SYSTEM INTERFACES

Leafcell systems cannot be distinguished from compound systems when viewed from the ´outside´, both share the same interfacing constructs.

<system_interface_part> ::= "system_name" <formal_terminal_part> ";"
[<formal_parameter_part> ";"] .

The system name introduces the systems template name. The template name is used as an reference to a system definition when it is used as an instance in a compound system. The template name is global, two templates with identical names cannot be used even when they are contained in separate files.

5.1 Terminal Declarations

A terminal port or terminal for short can be visualized as a contact at the perimeter of a system. Terminals constitute the interface of a circuit with its environment (neighbouring circuits). Each terminal implies a net with the same name, this net can be used to connect the system with other parts of its environment.

<formal_terminal_part> ::=
"(" [<terminal_declaration> { ";" <terminal_declaration> }] ")" .

Terminals are optional, some systems have no external connections. These systems generate all the signals that are needed for their behaviour internally. The braces ("(" and ")") however are mandatory.

<terminal_declaration> ::= <terminal_list> ":" <attribute_list> .

Each terminal declaration associates one or more terminals with a number of attributes.

5.1.1 Terminals

Terminals which share the same attributes can be assembled in a terminal list.

<terminal_list> ::= <terminal> { "," <terminal> } .

Let **T1, T2, .. Tn** be a collection of terminals, and **A** an instance of an attribute list associated with these terminals then,

T1, T2, .. Tn : A

is semantically equivalent with

T1 : A;
T2 : A;
..
Tn : A

Designers of digital circuits often use bundles of nets called ´busses´ or ´arrays´ to express their functional grouping. A vectorized terminal serves a common purpose, a number of terminals share the same name but have different indices. Two-dimensional terminals can be thought of as connectors, and the reader might find applications for even more dimensions.

<terminal> ::= "terminal_name" [<dimension_specification>] .

<dimension_specification> ::= "[" <expression> { "," <expression> } "]" .

The dimension specification specifies the number of terminals associated with a single name. The implementation of the NDML++ compiler imposes no restrictions on the number of dimensions a terminal can have.

Each expression specifying the terminals array size for a certain dimension index must evaluate to an integer value. The expressions can contain parameters (discussed in the next paragraph), this means that terminal dimensions can vary depending on the parameters supplied for a certain instance.

All terminal arrays have 0-origin, for example the terminal declared A[3] consists of the elements A[0], A[1], and A[2]. Multi-dimensional terminal arrays are expanded such that the last subscript varies most rapidly. That is the terminal array B[2,3] expands to B[0,0], B[0,1], B[0,2], B[1,0], B[1,1], and B[1,2]. Signal data is visualized to pass through a terminal, among things the terminal attributes describe this signal. Attributes are not essential from a network point of view, therefore they are considered not to belong to the core of the description language. Attributes are a means to enforce particular connection rules that assist the Eindhoven Piece-wise linear circuit simulator in verifying the correctness of his description.

<attribute_list> ::= "attribute_name" { "," "attribute_name" } .

Attributes are not reserved words, however a number of them have special (reserved) meaning for the Eindhoven Piece-wise linear circuit simulator.

Attribute	*Usage*
electr	The simulation program determines a voltage at, and a current through the terminal.
signal	The simulation program determines a single value to be passed from one terminal to another.
flow	The simulation program determines a flow value through the terminal.
input	The terminal is assumed only to read values from the environment.
output	The terminal is assumed only to write values to the environment.
inout	The terminal can write and read values to/from the environment.
unused	The terminal does not affect the internal behaviour of the system.

Signals behave similar to voltages in a sense that a single value is distributed between connected terminals. However signal terminals have no associated current. On the other hand flows behave similar to currents in the sense that all connected currents and flows are summed to zero. The electrical, signal, and flow attributes are mutually exclusive on a single terminal.

The input, output, inout, and unused attributes are also mutually exclusive.

The above attributes are treated as names and consequently are case sensitive.

Examples of terminal declarations:

in[8] : in, signal

ground, Vcc : electr

connector[2,14] : inout, signal

5.2 Formal Parameters

System definitions may be parameterised. Parameters allow the behaviour of a system to be altered upon instantiation. Formal parameter declarations are an optional part of the interface of a system definition. At evaluation time, the actual parameter values are obtained by evaluation of expressions. At simulation time, parameter values can be considered as constants.

<formal_parameter_part> ::=
"<<" <parameter_declaration> { ";" <parameter_declaration> } ">>" .

Parameters are introduced by means of parameter declarations. All parameters must be declared here, the NDML++ language has no internal parameters. As an substitute for an internal parameter, a parameter with default value can be used.

<parameter_declaration> ::= <parameter_list> [<type_and_default_specification>] .

<parameter_list> ::= "parameter_name" { "," "parameter_name" } .

<type_and_default_specification> ::= ":" (<default_value_specification>
| <value_range_specification> [<default_value_specification>]) .

Parameters which share the same type and default specification can be assembled in a parameter list.

Let **P1, P2, ... Pn** be an instance of a parameter list and **TD** an instance of the type and default specification, then

P1, P2, ... Pn TD

is semantically equivalent with

P1 TD ;
P2 TD ;
..
Pn TD

5.2.1 Value Range Specification

A value-range-specification states that the value of the parameter must always lie within or be equal to the specified boundary values.

<value_range_specification> ::= <bound> ".." <bound> .

<bound> ::= ["-"] <constant>
| <no_bound_denoter> .

<no_bound_denoter> ::= "" .*

The bounds must be constants optionally preceded by a minus sign, where the lower-bound must be smaller or equal to the upper-bound. A no-bound-denoter specifies a one-sided open interval of values.

If no value_range_specification is present in a parameter declaration then it is assumed that the parameter may take any number literal as its value. This is equivalent to "* .. *" .

5.2.2 Default Value Specification

The default_value_specification specifies a default value for the parameter.

<default_value_specification> ::= "DEFAULT" "-" <constant> .

The default value must be a legal value for the parameter, the default value type and boundary value types (if present) must be equal.

Examples of parameter declarations:

amplitude : 0.0 .. * DEFAULT 1.0

v_init, i_init : DEFAULT 0.0

output_buffer : FALSE .. TRUE DEFAULT FALSE

required_parameter

6. LEAFCELL SYSTEMS

A leafcell system (or leafcell for short) models the behaviour of a certain hardware component by means of piecewise-linear equations augmented with first-order differential equations. The latter type of equations allow for dynamic, i.e. time-dependent behaviour description. Ordinary linear equations can be seen as a special case of piecewise-linear equations, however in NDML++ this distinction is made explicit, so in general a leafcell contains these three types of equations.

More conventional circuit simulators ([SPICE], [ASTAP]) typically start out from a set of linear, non-linear and differential equations, which they solve using well-known numerical methods. It is clear that compared with a piecewise-linear simulator the difference lies in the treatment of non-linearity.

As stated earlier, leafcell systems cannot be distinguished from compound systems when viewed from the 'outside', both share the same interfacing constructs. The difference in syntax between both kind of systems becomes apparent when we look at the body of a leafcell definition. The syntax for the body of a leafcell directly reflects the mathematical notation commonly in use to describe piecewise-linear equations ([Bokhoven], [Eijndhoven]). All equations can conveniently be described in matrix-form. Furthermore some programming-like constructs are added that provide the user with an easy construction method for leafcells that have a variable number of terminals.

<leafcell_definition> ::=
"LEAFCELL" <system_interface_part> <leafcell_implementation> .

<leafcell_implementation> ::= <matrix_definition> | <select_statement> .

The leafcell keyword signals the compiler that we are dealing with a leafcell system definition.

The system interface part of a leafcell system definition describes those aspects of the system that need to be known to the 'outside world'. The interface part introduces a name for the system template and declares the formal terminal ports and formal parameters. See chapter 4 for a complete discussion of the syntax.

6.1 The Select Statement

In general, a select-statement will be used to bring related sets of equations in one and the same leafcell definition. Here 'related' means that the matrices describe basically the same kind of behavioural model but with different modeling concepts. As an example, consider a leafcell used to describe the behavioural aspects of a bipolair transistor. Several models with different aspects can be used, for instance the Gummel-Poon or Ebers Moll model. Upon instantiation a parameter can be set to choose the required behaviour.

<select-statement> ::= <clause> { <clause> } [<otherwise-clause>] .

<clause> ::= <guard> <matrix-definition> .

<guard> ::= "CASE" <expression> .

<otherwise-clause> ::= "OTHERWISE" <matrix-definition> .

A select-statement offers at least one case, each case consisting of a matrix-definition preceded by a guard. The otherwise-clause is optional but when it appears it must be the last case. The guard is introduced by the reserved word "CASE" and states a condition by means of an expression that evaluates to a boolean value. The select-statement states that the applicable matrix-definition is the one with a guard that evaluates to true. (Note the difference in semantics compared to the Pascal case statement.) When none of the guards is true and an otherwise clause is present then this matrix-definition applies. The compiler will generate an error message if none of the guards evaluates to

"TRUE", and there is no otherwise clause specified. Also the compiler will warn against multiple guards that evaluate to "TRUE".

6.2 The Matrix Definition

The leafcell statements construct the matrix definition that describes the equations in matrix form. The matrix rows are labeled by a keyword indicating the type of equation, the columns are labeled with their corresponding variable.

The language provide some programming like constructs that permit the conditional and iterative manipulation of matrix elements.

<matrix_definition> ::= "BEGIN" [<reference_definition>] <leafcell_statement_list> "END" .

<leafcell_statement_list> ::= <leafcell_statement> { ";" <leafcell_statement> } .

<leafcell_statement> ::= <variable_declaration>
/ <row>
/ <assignment_statement>
/ <remove_statement>
/ <pivot_statement>
/ <optional_leafcell_statement>
/ <leafcell_statement_repetition> .

6.3 The Voltage Reference

Terminals and nets of type "ELECTR" can be viewed to consists of a voltage component and a current component. When they appear as variables in leafcell equations this distinction must be explicitly made, moreover we must also specify a common 'ground' terminal, with respect to which all voltages are expressed within each leafcell. For that purpose the reference definition is introduced:

<reference-definition> ::= "REFERENCE" "=" <formal-terminal-name> ";" .

<formal_terminal> ::= "formal_terminal_name" [<index>] .

The formal-terminal-name must be of type "ELECTR". The electrical terminal port declared to be the reference may not appear as variable in the variable-list.

When the formal terminal is declared with an dimension specification the proper element must be selected using normal indexing.

6.4 The Variable Definitions

Parts of the matrix definition that contain many non-empty elements are described separately. Each part represents a number of columns and rows.

<variable_declaration> ::= "VAR" "(" <variable_list> ")" .

<variable_list> ::= <variable_identification> { "," <variable_identification> } .

<variable_identification> ::= <internal_variable_identification>
| <external_variable_identification>
| <source_vector_identification> .

<internal_variable_identification> ::= <variable_type> "." <expression> .

<variable_type> ::= "PL" | "U" .

<external_variable_identification> ::= <formal_terminal> ["." <extension>] .

<extension> ::= "I" | "V" .

<source_vector_identification> ::= <numeric_literal> .

The variable-list enumerates the variables that appear in the equations. Their position corresponds to the column of the matrix where their coefficients appear. The source vector identification denotes a constant column vector. All corresponding elements of the last matrix column are multiplied by this constant.

Two groups of variables are distinguished:

1. Variables denoting the signal value at one of the terminal ports of the leafcell system. This group is called the external variables. When this signal is of type "ELECTR" both its voltage component and its current component appear explicitly as separate variables and therefore appear with a proper extension to its name.
2. Variables introduced by the piecewise-linear modeling technique and variables used in the differential equations: the internal variables group. These are denoted by the reserved word "PL", c.q. "U", suffixed with a non-negative integer number, in order to distinguish each of them.

Example of an variable declaration:

VAR (out, u.1, pl.1, pl.2, 1);

6.5 The Matrix Rows

Each row in the matrix definition is an equation of a certain kind expressing a relation between some of the variables appearing in the last declaration of the variable-list. The kind of equation is determined by the row-identification, which starts with a reserved word. Each row element can be an expression resulting in a coefficient to be multiplied with the variable corresponding to the column position of the element. Elements may be left empty in which case the coefficient is taken to be zero.

<row-list> ::= *<row> { <row> } .*

<row> ::= *<row-identification> "=" <row-element-list> ";" .*

<row-identification> ::= *<equation-type> "." <expression> .*

<equation-type> ::= *"ZERO" | "DU" | "PL" .*

<row-element-list> ::= *<row-element> { "," <row-element> } .*

<row-element> ::= *[<expression>] .*

The three kinds of equations are:

1. Linear equations, indicated with the reserved word "ZERO" and followed by a suffix. In conventional mathematical terms such an equation reads:
   $0 = a_{i1} * \text{var1} + a_{i2} * \text{var2} \ldots$
2. Piecewise-linear equations, starting with the reserved word "PL" again followed by a suffix to properly distinguish between more of them. These equations relate the so-called complementary state-variables, often denoted with the letters "v" and "i" (see the literature about piecewise-linear modeling for more theoretical details). In PWL terms such an equation reads:
   $v_i = \ldots + d_{ij} * i_j + d_{ij+1} * i_{j+1} + \ldots$
3. First-order ordinary differential equations, indicated with "DU" followed by a suffix. Such an equation expresses the derivative of an "U" variable with respect to time (dU/dt) as a linear combination of some variables in the variable-list.

Example of a matrix row (see previous example for the corresponding variable declaration):

zero.1 = -1, , , 1, , ;

6.6 Assignment Statement

An assignment statement assigns a value to a matrix element. Any previously assigned value is lost. The matrix-element-identification specifies the element to which a new value is assigned. The value must be integer or real.

<assignment-statement> ::= <matrix-element-identification> ":=" <expression> .

<matrix-element-identification> ::=
"[" <row-identification> "," <variable-identification> "]" .

Example of an assignment statement:

```
[zero.1, D.i] := -Rd
```

6.7 Remove Statement

The action of a remove statement is to eliminate a variable from the matrix-definition. Only internal variables may be removed, it is an error otherwise. The variable identification specifies the column corresponding to the variable to be removed. The equation type should correspond to the variable type. Remove statements on off-diagonal elements are allowed for PL row and columns only, and should be used only when the diagonal element equals zero.

<remove-statement> ::= "REMOVE" <matrix-element-identification> .

6.8 Pivot Statement

The pivot statement is intimately bound to the algorithm used by the simulator. Its effect is to pivot a complementary PWL variable pair. This causes the simulator to start in a different linear segment. The matrix-element-identification must specify a "PL" type of row and "PL" type of variable (column).

<pivot-statement> ::= "PIVOT" <matrix-element-identification> .

6.9 Optional Leafcell Statement

If statements allow for conditional execution of sequences of other leafcell statements depending on the result of evaluating an expression.

<optional_leafcell_statement> ::= "IF" <expression> "THEN" <leafcell_statement_list>
["ELSE" <system_statement_list>] "FI" .

The expression must evaluate to a boolean value. If it evaluates to "TRUE" then control is passed to the leafcell statements following the "THEN", if on the other hand the expression evaluates to "FALSE" the leafcell statements in the "ELSE" clause if present are executed.

Example of an if-then statement:

```
IF init_Vg <Vfb THEN
  PIVOT[pl.1, pl.1];
ELSE
  PIVOT[pl.2, pl.2];
FI;
```

6.10 Iterative Leafcell Statement

The for-next statement allows the multiple execution of a leafcell statement sequence, a local parameter will count the number of iterations.

<leafcell_statement_repetition> ::= "FOR" <loop_condition> <leafcell_statement_list> "NEXT" .

<loop_condition> ::= "local_parameter_name" ":=" <expression> "UNTIL" <expression> .

The local parameter name is assigned the value the first expression evaluates to, this value must be integer. The leafcell statement(s) contained in the body of the for-next statement are executed and the local parameter value is incremented by "1". This continues until the second expression evaluates to "TRUE".

The local parameter value only has meaning inside the for-next loop. For-next statements can be nested, but the local parameter names must be distinct. The use of a local parameter name hides the definition of an actual parameter value, inside the for-next statement.

Examples of a for-next statement:

```
for j:=0 until j==n
    [zero.1, bit[j]] := 2**j;
next;
```

7. COMPOUND SYSTEMS

A compound system defines a template for the structure of a (sub-)network. It can be understood as a sort of macro definition for a certain interconnection pattern between certain components. The compound template then can be used to create instances (macro calls) in order to build a more complex network. Like with macros of an assembler language, compounds (also leafcells) may be parameterised to achieve more flexibility in using them.

<compound_system> ::=
"COMPOUND" <system_interface_part> <compound_implementation> ";".

The compound keyword signals the compiler that we are dealing with a compound system definition.

The system interface part of a compound system definition describes those aspects of the system that need to be known to the 'outside world'. The interface part introduces a name for the system template and declares the formal terminal ports and formal parameters. See the previous chapter for a complete discussion of the syntax.

<compound_implementation> ::=
<instance_declaration_part> [<net_declaration_part>] <compound_body> .

The implementation of the compound system contains the instances, the (optional) nets that are used to interconnect them, and the actual interconnection description.

7.1 Instance Declarations

The system instances constitute the structural components of the compound system, i.e. they specify the subsystems of which the system being defined is built of. A system instance is said to be derived of a system definition or template. A system instance cannot be derived from the system containing the declaration, i.e. recursive instantiation is not allowed.

<instance_declaration_part> ::= "INSTANCE" { <instance_declaration> ";" }+ .

<instance_declaration> ::=
[<instance_list> ":"] "template_name" [<actual_parameter_part>] .

Each instance declaration associates an instance list with the system template and a set of actual parameters. If the user chooses to name a single instance after the template name, the instance list and the ":" can be deleted.

Let **I** stand for some instance, **T** for some template name and **AP** is the actual parameter part (possibly empty), then

T AP

is semantically equivalent with

T : T AP

7.1.1 Instances

Instances which share the same template and actual parameters can be assembled in a instance list.

<instance_list> ::= <instance> { "," <instance> } .

Let **I1, I2, ... In** stand for some instance list, **T** for some template name and **AP** is the actual parameter

part (possibly empty), then

I1, I2, ... In : T AP

is semantically equivalent with

I1 : T AP ;
I2 : T AP ;
..
In : T AP

Instances can also be vectorized, a number of instances share the same instance name but have different indices. Examples of one dimensional instance arrays are registers, and transmission-lines. Two-dimensional instance arrays can be used for the construction of memories, PLA's etc.

<instance> ::= "instance_name" [<dimension_specification>] .

<dimension_specification> ::= "[" <expression> { "," <expression> } "]" .

The dimension specification specifies the number of instances associated with a single name. The implementation of the NDML++ compiler imposes no restrictions on the number of dimensions an instance can have.

Each expression specifying the instance array size for a certain dimension index must evaluate to an integer value. The expressions can contain parameters, this means that instance array dimensions can vary depending on the parameters supplied for a certain instance.

All instance arrays have 0-origin, for example the instance declared A[3] consists of the instances A[0], A[1], and A[2]. Multi-dimensional instance arrays are expanded such that the last subscript varies most rapidly. That is the instance array B[2,3] expands to B[0,0], B[0,1], B[0,2], B[1,0], B[1,1], and B[1,2].

7.1.2 Actual Parameters

Part of the declaration forms the association of values to the formal parameters of the template system. These values are called the actual parameters of the instance. Association of actual with formal parameters is done by means of parameter assignment statements. This approach is often referred to as named parameter association.

<actual_parameter_part> ::=
"<<" <parameter_assignment> { "," <parameter_assignment> } ">>" .

<parameter_assignment> ::= "formal_parameter_name" ":=" <actual_parameter> .

<actual_parameter> ::= <expression> .

The name appearing on the left-hand side of a parameter assignment must be the name of a formal parameter of the template system.

The expression must evaluate to a numeric value, and the value must be within the range if specified for the formal parameter.

If a system template specifies a default value for a parameter, then the parameter may be omitted from the actual parameter part, i.e. there need not be an assignment statement with the name of that parameter appearing on the left-hand side.

Examples of instance declarations:

FlipFlops[N] : MSFF (* N = width of register *)

gen : SINE_GENERATOR << ampl := 10, frequency := 2k >>

driver : nmost << L:=4u, W:=(4+fanout)*1u >>

7.2 Net Declarations

Terminal names can implicitly be used as net names. Explicit nets are introduced by the net declarations. Net names may not be equal to formal terminal names.

<net_declaration_part> ::= "NET" { <net_declaration> ";" }+ .

<net_declaration> ::= <net_list> ":" <attribute_list> .

Each net declaration associates one or more nets with a number of attributes.

7.2.1 Nets

Nets which share the same attributes can be assembled in a net list.

<net_list> ::= <net> { "," <net> } .

Let **N1, N2, ... Nn** be an instance of the net list and **A** an
instance of the attribute list, then

N1, N2, ... Nn : A

is semantically equivalent with

N1 : A ;
N2 : A ;
..
Nn : A

Designers of digital circuits often use bundles of nets called 'busses' or 'arrays' to express their functional grouping. A vectorized net serves a common purpose, a number of nets share the same name but have different indices.

<net> ::= "net_name" [<dimension_specification>] .

<dimension_specification> ::= "[" <expression> { "," <expression> } "]" .

The dimension specification specifies the number of nets associated with a single name. The implementation of the NDML++ compiler imposes no restrictions on the number of dimensions a net can have.

Each expression specifying the net array size for a certain dimension index must evaluate to an integer value. The expressions can contain parameters, this means that net dimensions can vary depending on the parameters supplied to the compound instantiation.

All net arrays have 0-origin, for example the net declared A[3] consists of the elements A[0], A[1], and A[2]. Multi-dimensional net arrays are expanded such that the last subscript varies most rapidly. That is the net array B[2,3] expands to B[0,0], B[0,1], B[0,2], B[1,0], B[1,1], and B[1,2].

7.2.2 Net Attributes

Among things the net attributes describe the type of data that 'flows' though the net. Attributes are not essential from a network point of view, therefore they are considered not to belong to the core of the description language. Attributes are a means to enforce particular behaviour that assist the Eindhoven Piece-wise linear simulator in verifying the correctness of the network.

<attribute_list> ::= { "attribute_name" { "," "attribute_name" } .

Attributes are not reserved words, however a number of them have special meaning for the Eindhoven Piece-wise linear simulator .

Attribute	*Usage*
electr	The simulation program determines a voltage at, and a current through the net .
signal	The simulation program determines a single value to be passed from one terminal to another.
flow	The simulation program determines a flow value through the terminal.

Signals behave similar to voltages in a sense that a single value is distributed between connected terminals. However signal nets have no associated current. On the other hand flows behave similar to currents in the sense that all connected currents and flows are summed to zero. The electrical, signal, and flow attributes are mutually exclusive on a single net.

The input, output, inout, and unused attributes have no meaning for nets.

The above attributes are treated as names and consequently are case sensitive.

Examples of net declarations:

internal_net, internal_connection : electr

data_bus[N] : signal

7.3 The Compound Body

The compound body defines the interconnection of the terminal ports of the declared instances, nets, and formal terminal ports of the system being defined. The language provides some programming like constructs that permit conditional and iterative interconnection.

<compound_body> ::= "BEGIN" <compound_statement_list> "END" .

<compound_statement_list> ::= { <compound_statement> ";" }+ .

<compound_statement> ::= <terminal_interconnect_statement>
| <instance_invocation_statement>
| <compound_statement_repetition>
| <optional_compound_statement> .

7.4 Instance Invocation

The system instance(s) declared previously must be connected to each other and the formal terminals of the current compound system. Optionally nets also previously declared can be used to accomplish or simplify this connection.

<instance_invocation_statement> ::= <system_instance> "(" <actual_net_list> ")" .

The actual nets will be connected in the same order as the definition of the formal terminals of the system that is invocated. Each actual formal terminal size and the corresponding actual net size must agree.

7.4.1 System Instances

The system instance must be declared by an instance declaration before it can be used, illegal use will be reported by the compiler. However instances that are declared but are not used are not noted by the compiler.

<system_instance> ::= "instance_name" [<index>] .

<index> ::= "[" <expression> { "," <expression> } "]" .

The index is optional, if an instance is declared with a dimension specification, indices must be used to distinguished instances from each other. A system instance invocation must be specified for each valid index of a certain instance. However the compiler does not check for this criterion. The compound repetition statement defined in one of the following paragraphs can be used to facilitate this.

The expressions for the individual indices must evaluate to an positive integer value. The compiler checks whether the indices specified are correct, and/or if an index is missing.

7.4.2 Actual Nets

The actual nets are connected to the formal terminals of the system instance.

<actual_net_list> ::= <actual_net> { "," <actual_net> } .

<actual_net> ::= "actual_net_name" [<subrange>]
| <bundle>
| "-" .

An actual net is either an previously declared net or a formal terminal of the current system.

If a formal terminal is not connected to its environments a "-" is put in the place of the formal terminal. The formal terminal left open can have any dimension possible, one "-" will suffice.

A subrange can be used to select multiple nets from a actual net declared with an dimension specification. The subrange determines the number of actual nets that are connected to the formal terminal.

<subrange> ::= "[" <subrange_list> "]" .

<subrange_list> ::= <subrange_item> { "," <subrange_item> } .

<subrange_item> ::= <expression> [".." <expression>] .

Each subrange item selects the valid indices for a certain dimension. An expression in the subrange item must evaluate to a positive integer value, furthermore it must be a valid value for the specific

actual net. If two expressions are specified for a subrange item all indices in between and including these indices are selected. The actual net size depends on the selected subrange, for instance A[0..1,0..3] has size [2,4]. On the other hand A[1,1] has 0-dimension and size '1'. A single subrange item is used to select an actual net that has been declared with a dimension specification.

Multiple actual nets can be bundled to a single net with the bundle constructors "<|", "|>", and "<_", "_>".

<bundle> ::= "<_" <actual_net_list> "_>"
| "<_" <actual_net_catenation> "_>"
| "<|" <actual_net_list> "|>"
| "<|" <actual_net_catenation> "|>" .

<actual_net_catenation> ::= <actual_net> { "|" <actual_net> } .

The elements of an actual net list are stacked on each other, each element must have the same size. An unconnected net ("-") can be used, and will inherit the same size. The result of this operation is a new actual net that has one additional dimension. When the "<_" and "_>" bundle constructors are used on an actual net list the actual nets are stacked on each other in the additional (last) dimension. For instance if three two-by-two actual nets are stacked with the previously mentioned bundle constructors, a two-by-two-by-three actual net results. On the other hand when the "<|" and "|>" bundle constructors are used on an actual net list the actual nets are stacked on each other in the first dimension. Use of these bundle constructors on three two-by-two actual nets will result in a three-by-two-by-two actual net.

The elements of the actual net catenation may only differ in the first or last dimension, depending on the type of bundle constructors used. An unconnected net ("-") can be used, and will inherit the same size. The "<_" and "_>" bundle constructors catenate a number of actual nets in the last dimension, all other dimensions must agree. An actual net results that has equal size in all but its last dimension. For this dimension the sizes of all actual nets are summed. For instance a two-by-two, and a two-by-three actual net catenated this way will result in a two-by-five actual net. As can be expected the "<|" and "|>" bundle constructors catenate a number of actual nets in the first dimension. An actual net results that has equal size in all but its first dimension. For this dimension the sizes of all actual nets are summed.

Examples of instance invocation statements:

resistor(in,out)

MSFlipFlop(clock,set,reset,output,-)

PLA(<_a, b, c, d, e, f_>,<_display_1[0..6], display_2[0..6]_>)
(* this system could be defined as PLA(x[5], y[2,7]) *)

7.5 Net Connection

Internal connection of actual nets can be accomplished with the terminal interconnect statement.

<terminal_interconnect_statement> ::= "CONNECT" "(" <actual_net_list> ")" .

The actual nets that are connected must have equal sizes.

The definition and construction of actual nets are described in the previous paragraph.

Examples of a connect statement:

```
CONNECT(in, out)

CONNECT(<| add[0..2], sub[0..2] |>, out[0..1,0..2])
```

7.6 Optional Compound Statement

If statements allow for conditional execution of sequences of other statements depending on the result of evaluating an expression.

<optional_compound_statement> ::= "IF" <expression> "THEN" <compound_statement_list>
["ELSE" <compound_statement_list>] "FI" .

The expression must evaluate to a boolean value. If it evaluates to "TRUE" then control is passed to the compound statements following the "THEN", if on the other hand the expression evaluates to "FALSE" the compound statements in the "ELSE" clause if present are executed.

Example of an if-then statement:

```
IF output_buffer==TRUE THEN
   buffer(out,output);
ELSE
   connect(out,output);
FI;
```

7.7 Iterative Compound Statement

The for-next statement allows the multiple execution of a compound statement sequence, a local parameter will count the number of iterations.

<compound_statement_repetition> ::= "FOR" <loop_condition> <compound_statement_list> "NEXT" .

<loop_condition> ::= "local_parameter_name" ":=" <expression> "UNTIL" <expression> .

The local parameter name is assigned the value the first expression evaluates to, this value must be integer. The compound statement(s) contained in the body of the for-next statement are executed and the local parameter value is incremented by "1". This continues until the second expression evaluates to "TRUE".

The local parameter value only has meaning inside the for-next loop. For-next statements can be nested, but the local parameter names must be distinct. The use of a local parameter name hides the definition of an actual parameter value, inside the for-next statement.

Examples of a nested for-next statement:

```
FOR p:=0 UNTIL p==(N-1)
  FOR q:=0 UNTIL q==(M-1)
    ramcell(in[p],out[p],address[q]);
  NEXT;
NEXT;
```

8. NDML language evolution

A short overview is presented of the extensions to NDML with respect to earlier versions, and incompatibilities with the last version. Also plans for near future extensions and modification are mentioned.

8.1 Recent NDML extensions

The most important extensions from the latest NDML implementation are:

- The FOR-UNTIL-NEXT repetition statement and the IF-THEN-ELSE-FI selection statement. In leafcell bodies these statements allow the matrix structure and size to depend upon the leafcell parameters in a very flexible way. In compound bodies these statements allow to parameterize the network **structure** which was previously impossible.
- The array and index constructs for terminals, local nets and instantiations. They improve the readability of the language, and can strongly improve the compactness of a network description, specially in combination with the repetition statement.
- The bundle constructs to gather individual nets into an array form, required to connect to a conformable sized terminal.
- The support for integer expressions after the dot "." in leafcell matrix row and column identifications. Previously these needed to be integer constants.
- The include and library mechanism implemented by the NDML program, which greatly assists the user, but basically is not part of the NDML language.

8.2 Incompatibilities with previous NDML

The incompatibilities with the last release of NDML, as it was built into the Eindhoven Piecewise Linear Simulator, are:

- The operator that tests for equality was previously "=", and is now modified to "==". This was required to obtain a conflict free grammar, because expressions are now supported at more locations.
- The alternatives in a CASE statement to define a leafcell body were previously terminated by a semicolon ";". In this release the semicolon is not allowed between the alternatives, but must appear after the last one to terminate the CASE statement (as any other statement).
- New reserved words are added (FOR, UNTIL, NEXT, I, V, LOG2, LOG10, ARCCOS, ARCSIN, ARCTAN, MAX, MIN, ROUND, INITIAL, TRUE, FALSE) which might interfere with names in older ndml files.
- The empty statement is not allowed any more.

8.3 Plans for future extensions

The following (near-) future extensions and modifications to the NDML language are foreseen:

- Addition of local variables that can be assigned by an expression, and can be used in other expressions. This can considerably ease the definition of complex leafcells.
- Reduction of the set of reserved words, for instance by using normal (unreserved) names for functions.
- Allow for net and terminal attributes that can be assigned a numerical value. The main objective is to enable the simulation program to create capacitive loads on nets, by adding terminal capacitances.
- Support for the "INITIAL" keyword in leafcell definitions, to initialize u- or du- variables prior to the DC analysis phase of the simulator. By now this can only be accomplished in an indirect way.
- Allow a user to express a desire to numerically combine a compound system into a single leafcell matrix, instead of maintaining an interconnection between several (small) leafcell matrices. For often used small subnetworks this can bring a significant simulation speed up.

9. REFERENCES

[ASTAP]
Advanced Statistical Analysis Program (ASTAP),
Program Reference Manual, Pub. no. SH20-1118-0,
IBM Corp. Data Proc. Div., White Plains, NY 10604

[BDL]
Slutz, E., Okita, G., Wiseman, J.,
"Block Description Language (BDL): A Structural Description Language",
ACM IEEE 21-th Design Automation Conference Proceedings,
pp. 81-85, June 1984

[Bokhoven]
Bokhoven, W.M.G. van,
"Piecewise-Linear Modelling and Analysis",
Ph.D. dissertation, Eindhoven University of Technology, 1981

[Eijndhoven]
Eijndhoven, J.T.J van,
"A Piecewise Linear Simulator for Large Scale Intergrated Circuits",
Ph.D. dissertation, Eindhoven University of Technology, 1984

[Janssen]
Janssen, G.L.J.M.,
"Network Description & Modeling Language"
"The Integrated Circuit Design Book" pp. 4.60-4.108
Delft University Press 1986

[Graaf]
Graaf A.C. & Janssen, G.L.J.M.,
"Proposal for a Network Description Format in the ICD-System"
"The Integrated Circuit Design Book" pp.1.49-1.60
Delft University Press 1986

[Ruehli]
Chapter 1 "Circuit Description" in
"Circuit Analysis, Simulation and Design - Part I", A. E. Ruehli ed.
to be published by North-Holland

[SPICE]
Vladimirescu, A., Zhang, K., Newton, A.R., Pederson, D.O.,
Sangiovanni-Vincentelli, A.,
"SPICE Version 2G User's Guide",
Department of Electrical Engineering and Computer Sciences,
University of California, Berkeley, Augustus 1981

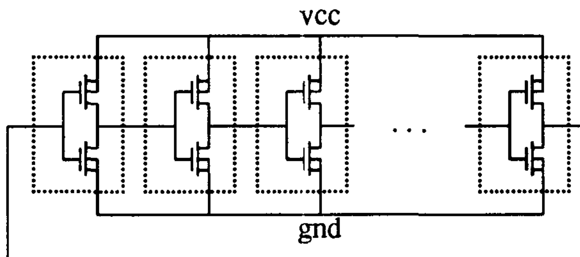
[Wirth]
Wirth, N.,
"What Can We Do about the Unnecessary Diversity of Notation
for Syntactic Definitions",
Comm. ACM, Vol. 20, Nr. 11, pp. 822-823, November 1977

APPENDIX 1: NDML++ EXAMPLES

This chapter presents a few examples of both compounds and leafcells, with their definition and their instantiation.

1.1 A CMOS ring oscillator

The circuit to be described in NDML++ is depicted below:



The corresponding network description could be:

```
compound ringosc(gnd, vcc : electr);
     << n : 1 .. * default 7>>;
instance inv[n] : inverter <<fanout := 1>>;
net    out[n] : electr;
begin
     for j := 0 until j == n
          inv[j](out[j], out[(j+1)%n], gnd, vcc);
     next;
end;

compound inverter( in, out, gnd, vdd :electr);
     <<fanout: 1 .. * default 3>>;
instance
     n : nmost << W := fanout * 4u >>;
     p : pmost << W := fanout * 4u >>;
begin
     n( in, out, gnd, gnd);
     p( in, out, vdd, vdd);
end;
```

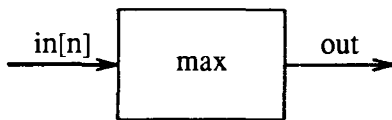
In the above example the compounds **ringosc** and **inverter** are specified in a single ndml++ input file. The file contains references to two modules (**nmost** and **pmost**) which are not defined. Because explicit references to include files are missing, the ndml++ parser will scan libraries to resolve these two references, as explained in section 3.2.

The modulo operator '%' in the index of the net **out**[] is responsible for the cyclic interconnect between the **n** inverters. Note that the compound **inverter** has no local nets: all nets are passed as argument and the **net** declarations can be omitted.

The parameter **fanout** of the compound **inverter** has default value 3, which is overridden by the assignment to 1 at instantiation. The resulting value is used in an expression, to assign the width **W** of the mos transistors. The constant **4u** is numerically equivalent to 4e-6 as explained in section 2.2.4.

1.2 A leafcell max

A matrix definition for a leafcell is presented, which determines at its single output the maximum value found in a vector of inputs.



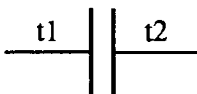
The corresponding ndml++ language definition is:

```
leafcell max_n( in[n]: signal, input; out: signal, output);
     << n: 1 .. * default 2>>;
begin
     var(  in[0], out);
     zero.1 =   1,  -1 ;
     for k:=1 until k == n
          var(  in[0], in[k], pl.1);
          pl.k =    1,   -1  ,  1  ;
          [zero.1, pl.k] := 1;
          for j:=1 until j == k
               [pl.k, pl.j+1] := 1;
          next;
     next;
end;
```

The terminal declaration defines **in** as a vector of width **n**, which allows indices 0 .. n-1. Two nested **for** loops are shown to define the matrix elements. The resulting matrix size implicitely follows from the assigned matrix elements. All elements not mentioned are implicitly set to zero. Two different ways of assigning are used here: either by a **var** statement followed by a set of row assignments (here only one), or assigning single matrix entries with [..] :=, see section 6.4 to 6.6.

1.3 A leafcell capacitor

A capacitor model is presented to show voltage and current equations in leafcells:



The corresponding matrix definition is:

```
leafcell capacitor( t1, t2: electr);
     << c: 0 .. *;
       r_series, leak_cond : 0 .. * default 0;
     >>;
begin
     reference = t2;
     var(    t1.v,  t1.i,      u.1);
     zero.1 =   -1,  r_series,    1 ;
     du.1   =      ,  1/c, -leak_cond/c;
end;
```

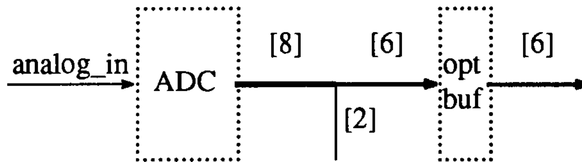
The model has parameters for the capacitance value (**c**), for an optional series resistance, and an optional leakage conductance. No default value for **c** is given, which makes a parameter assignment at instantiation mandatory.

The **reference** statement specifies **t2** as reference terminal for the local matrix definition only: The voltage level of all other terminals can appear as matrix column, and are taken with respect to this reference terminal. Furthermore the current through the reference terminal cannot appear as matrix column: This current is implicitly assigned, summming all terminal currents to zero.

Each **electr** typed terminal induces two column variables: a terminal voltage with extension **.v**, and a terminal current with extension **.i**. Incoming currents have positive sign.

1.4 An AD converter circuit

To show some more vectorised net constructs the following circuit is used:



The analog-to-digital converter creates an 8 bit output from its analog input value. However only the six most significant bits are used, the other two are left open. These six bits are passed to a buffer, which is in the ndml++ description only optionally inserted.

```
compound adc_with_opt_buf( in, clk: signal, input; out[6]: signal, output);
     << buf: default TRUE;
     >>;
instance
     adc: ad_converter;
     buf: latch <<n := 6>>;
net   bits[6]: signal;
begin
     adc( clk, in, <_ bits[0..5] | - | - _>);
     if buf
     then      buf( clk, bits[0..5], out[0..5]);
     else      connect( bits[0..5], out[0..5]);
     fi;
end;
```

The above bundle constructs with a <_ _> bracket pair catenate two unnamed nets '-' to a vector of length 6 (**bits[]**), resulting in a vector of length 8, which is needed to match the eight bit output vector of the adc module (see section 7.4.2).

Alternatively and more complicated, the instantiation of **adc()** could be specified as:

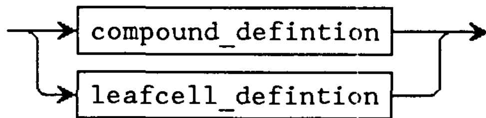
```
adc( clk, in, <_ bits[0..5] | <_-,-_> _>);
```

A vector is formed of two unnamed (scalar) nets, which is catenated to the vector of six nets of **bits**[]. Basically the bundle constructs allow the creation of any (multi dimensional) actual net from smaller structures, to match a given terminal.

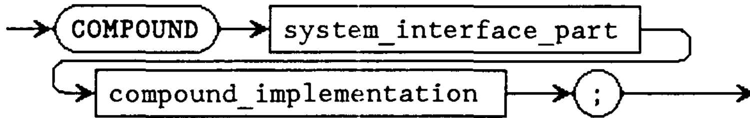
With an **if-then-else-fi** construct the latch compound is only conditionally inserted. The connect statements takes a list of actual nets of any equal size, and creates an element by element interconnect for each index (see section 7.5).

APPENDIX 2: NDML++ SYNTAX DIAGRAMS

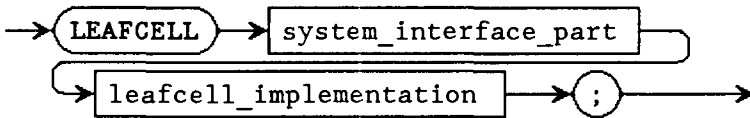
system_definition :



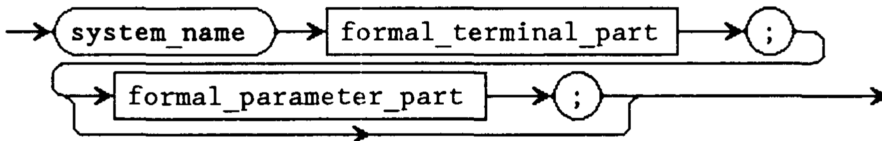
compound_definition :



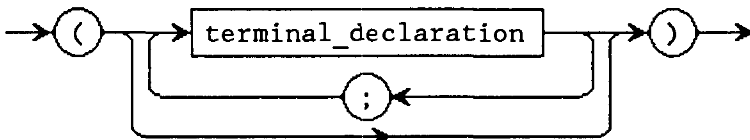
leafcell_definition :



system_interface_part :

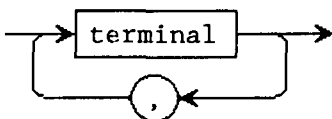


formal_terminal_part :

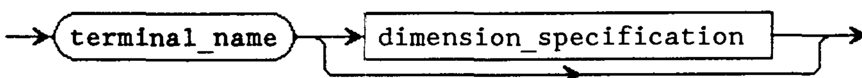


terminal_declaration :

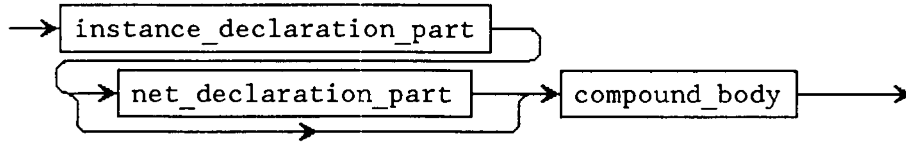
terminal_list :



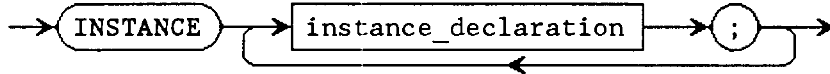
terminal :



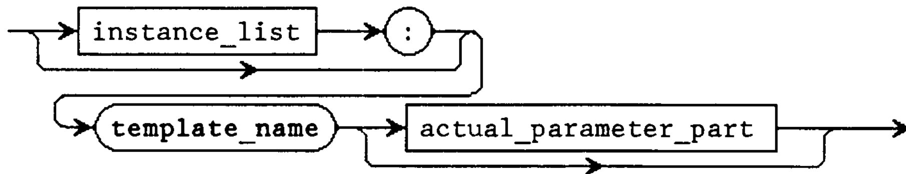
compound_implementation :



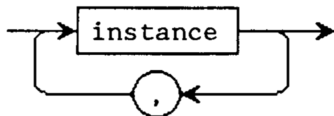
instance_declaration_part :



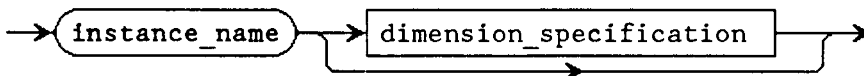
instance_declaration :



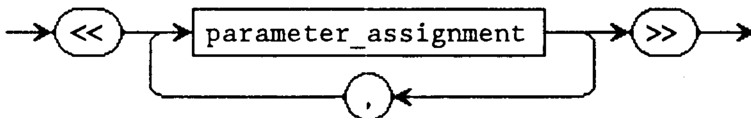
instance_list :



instance :

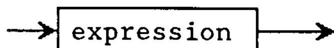


actual_parameter_part :

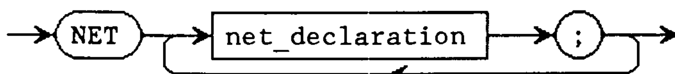


parameter_assignment :

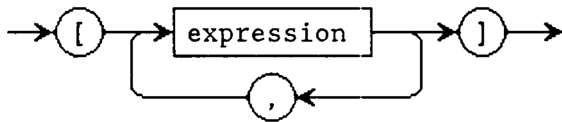
actual_parameter :



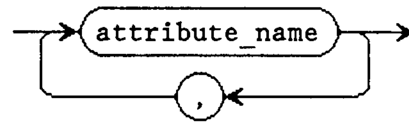
net_declaration_part :



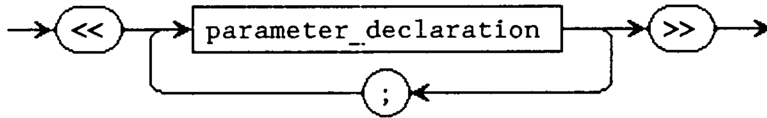
dimension_specification :



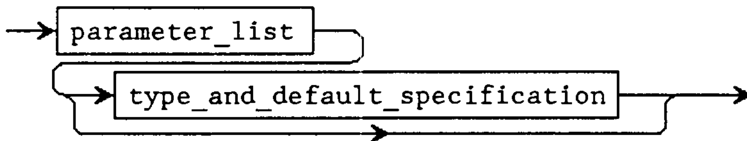
attribute_list :



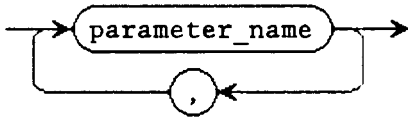
formal_parameter_part :



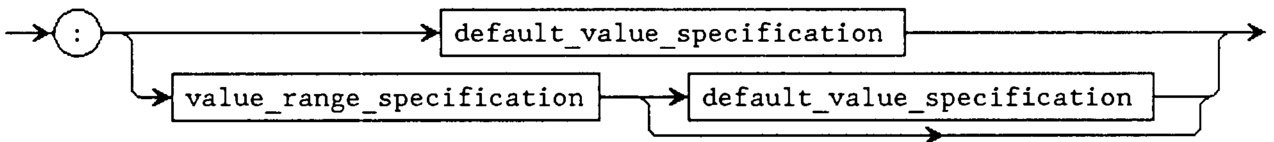
parameter_declaration :



parameter_list :

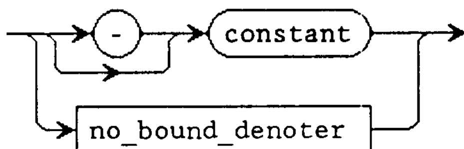


type_and_default_specification :



value_range_specification :

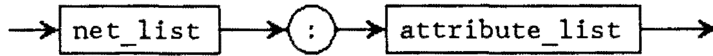
bound :



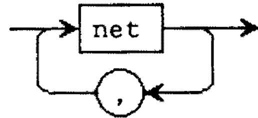
no_bound_denoter :

default_value_specification :

net_declaration :



net_list :

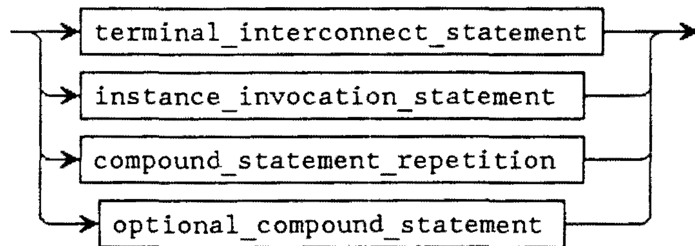


net :

compound_body :

compound_statement_list :

compound_statement :

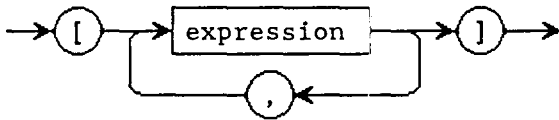


terminal_interconnect_statement :

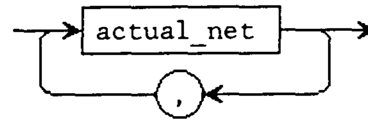
instance_invocation_statement :

system_instance :

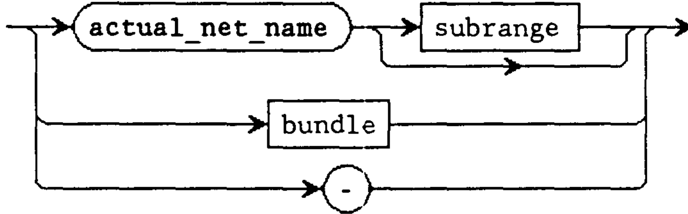
index :



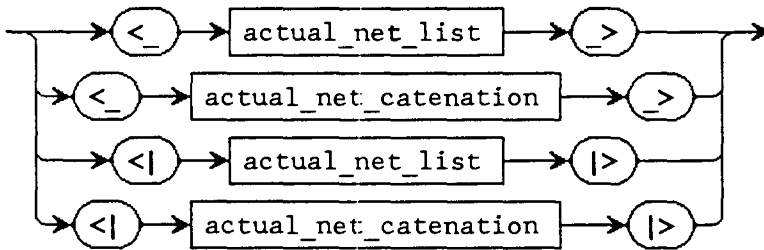
actual_net_list :



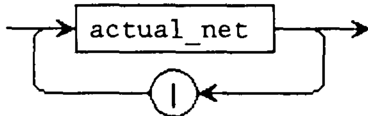
actual_net :



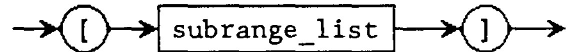
bundle :



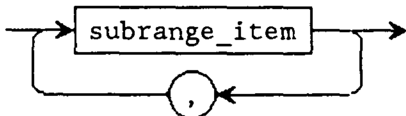
actual_net_catenation :



subrange :

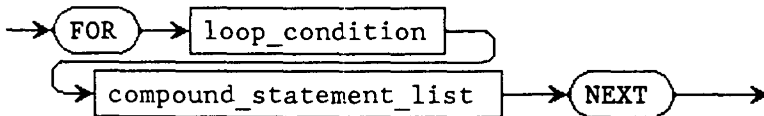


subrange_list :

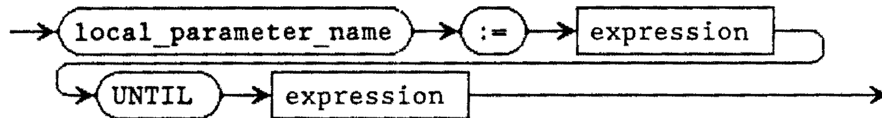


subrange_item :

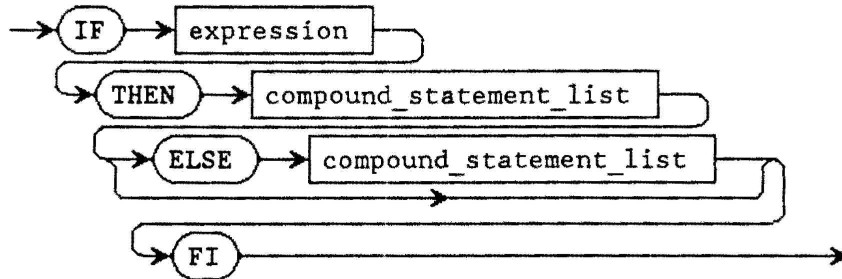
compound_statement_repetition :



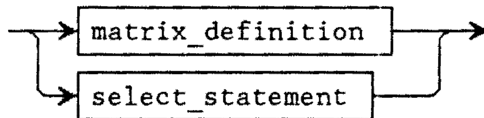
loop_condition :



optional_compound_statement :



leafcell_implementation :

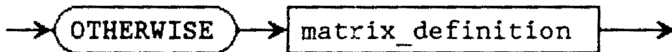


select_statement :

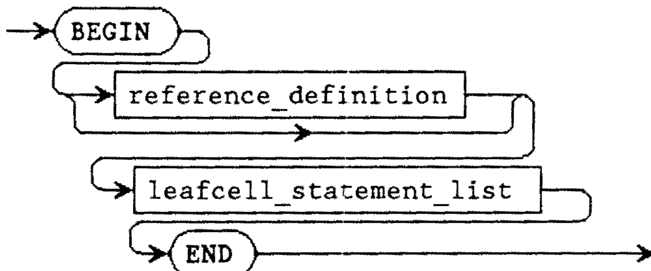
clause :

guard :

otherwise_clause :

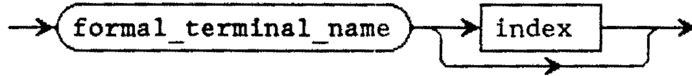


matrix_definition :

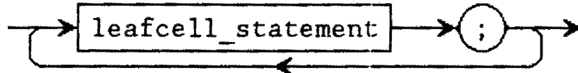


reference_definition :

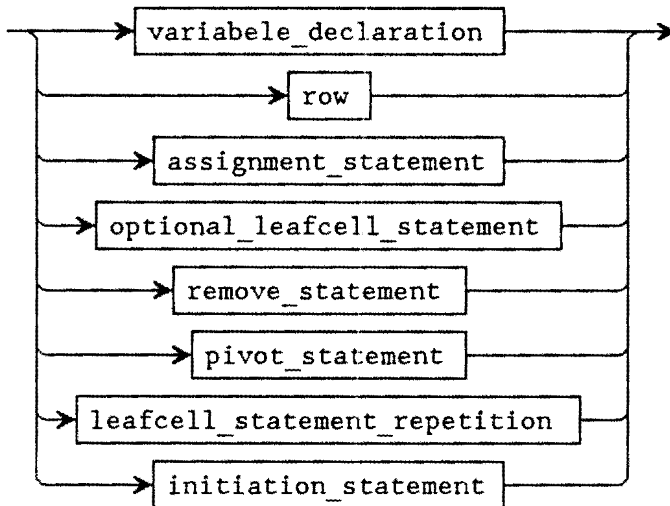
formal_terminal :



leafcell_statement_list :

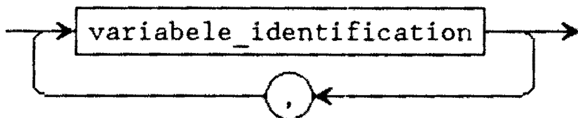


leafcell_statement :

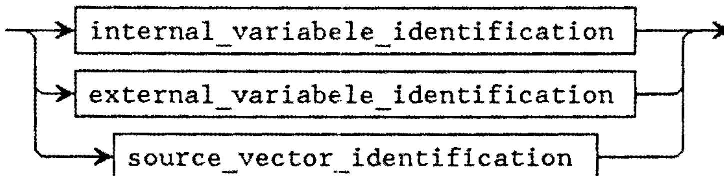


variabele_declaration :

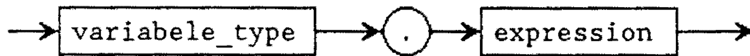
variabele_list :



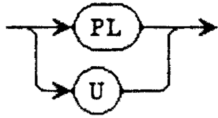
variabele_identification :



internal_variabele_identification :

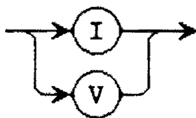


variabele_type :



external_variabele_identification :

extention :

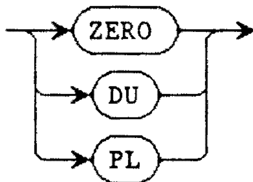


source_vector_identification :

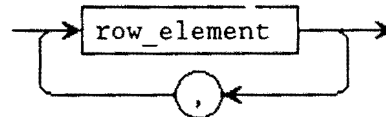
row :

row_identification :

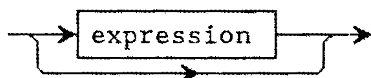
equation_type :



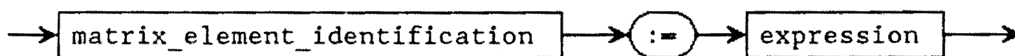
row_element_list :



row_element :

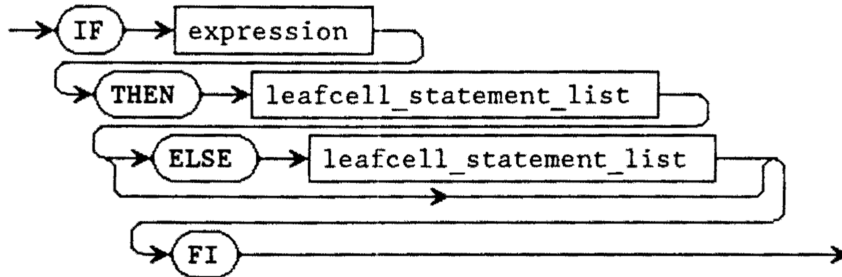


assignment_statement :

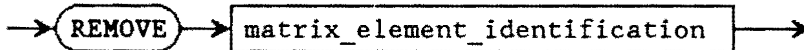


matrix_element_identification :

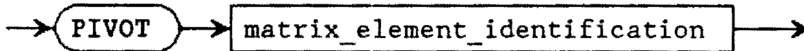
optional_leafcell_statement :



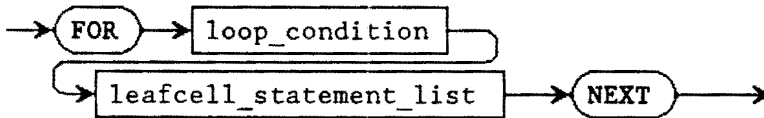
remove_statement :



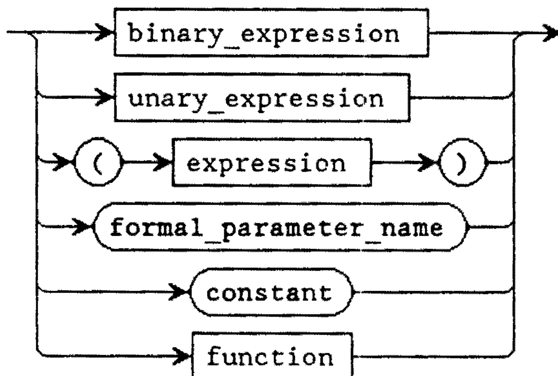
pivot_statement :



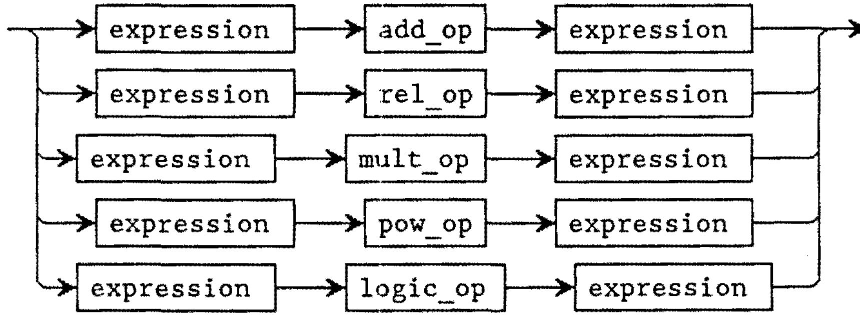
leafcell_statement_repetition :



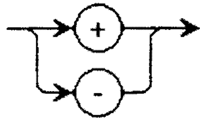
expression :



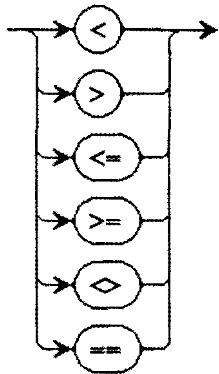
binary_expression :



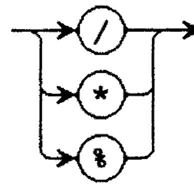
add_op :



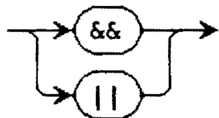
rel_op :



mult_op :

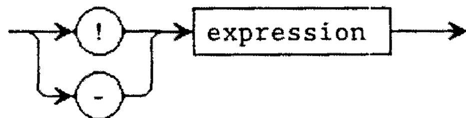


logic_op :

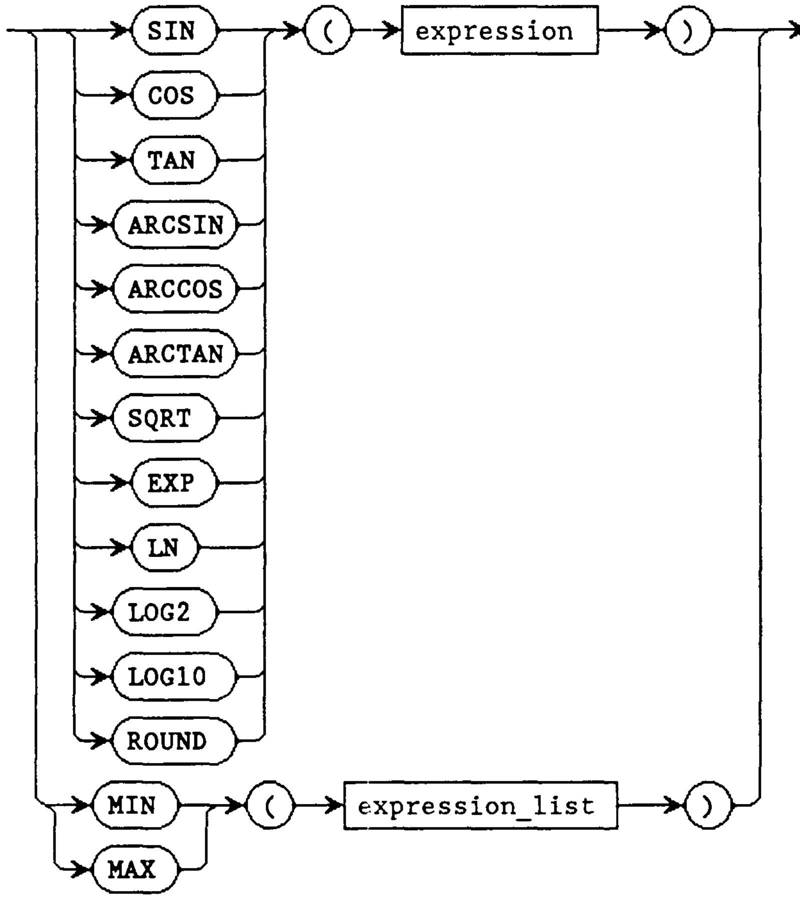


pow_op :

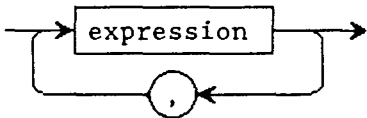
unary_expression :



function :



expression_list :



Index

A

Actual Net, 28, 29
  Bundles, 29
  Catenation, 29
  Subrange, 28
Actual Parameter, 25
ARCCOS Function, 13
ARCSIN Function, 13
ARCTAN Function, 13
Assignment Statement, 22
ASTAP, 18
Attribute, 32
  Net, 27
  Terminal, 15

B

BNF, 3
Boolean Literals, 7
Boundaries, 17
Bundles, 29

C

Character Case, 6
Character Set, 5
Comments, 8
Compatibility, 31
Compounds, 24
Constants, 7
COS Function, 13

D

Declaration:
  Instance, 24
  Net, 26
  Parameter, 16
  Terminal, 14
Default Value Specification, 17
Description Hierarchy, 10
Differential Equations, 18
Dimension Specification, 28
  Instance, 25
  Net, 26
  Terminal, 15

E

EBNF, 3
Equations, 19, 21
Errors:
  Classification of, 4
ESCHER Database, 10
ESCHER, 2
EXP Function, 13
Exponents, 7
Expressions, 11, 32
  Binary, 11
  Unary, 12
Extensions, 9, 31

F

Files, 10
Flows, 15, 27
Formal Terminal, 28
  Unconnected, 28
Functions, 13
Future Extensions, 32

G

Guard, 18

H

Hierarchy, 10

I

ICD Format, 10
Include Files, 10
Incompatibility, 31
index, 28
Initial, 32
Instance:
  Declaration, 24
  Dimension Specification, 25
  Invocation Statement, 28
  List, 24
  System, 28
  Vectorized, 25
Integer Literals, 7
Iterative Compound Statement, 30
Iterative Leafcell Statement, 23

L

Leafcells, 18
Lexical Elements, 5
Library Files, 10
LN Function, 13
Local Parameter, 23, 30

LOG10 Function, 13
LOG2 Function, 13

M

Matrix Rows, 21
MAX Function, 13
MIN Function, 13

N

Names, 6
Nested Comments, 8
Net Connect Statement, 29
Net:
  Actual, 28
  Attribute, 27
  Declaration, 26
  Dimension Specification, 26
  List, 26
  Vectorized, 26
No-bound-denoter, 17
Numeric Literals, 7

O

Operators, 6
  Additive, 11
  Boolean, 12
  Logical, 12
  Logical Not, 12
  Multiplicative, 11
  Precence & Associativity, 12
  Relational, 12
  Unary Minus, 12
Optional Compound Statement, 30
Optional Leafcell Statement, 22
Otherwise Clause, 18

P

Parameter:
  Actual, 25
  Assignment, 25
  Declaration, 16
  Default Value, 25
  Default Value Specification, 17
  Formal, 16
  Introduction, 16
  Local, 23, 30
  Value Range, 25
  Value Range Specification, 17
Pivot Statement, 22
PLATO, 2

R

Real Literals, 7
Remove Statement, 22
Reserved Words, 7, 32
ROUND Function, 13
Row:
  Elements, 21
  Identifcation, 21
  List, 21

S

Scale Factors, 7
Select Statement, 18
Separators, 6
Signals, 15, 27
SIN Function, 13
Spacing Conventions, 6
SPICE, 18
SQRT Function, 13
Statement:
  Assignment, 22
  For-next, 23, 30
  If-then, 22, 30
  Include, 10
  Instance Invocation, 28
  Iterative Compound, 30
  Iterative Leafcell, 23
  Net Connect, 29
  Optional Compound, 30
  Optional Leafcell, 22
  Pivot, 22
  Remove, 22
  Select, 18
Subrange, 28
  Item, 28
  List, 28
Syntax Notation, 3
System:
  Instance, 28
  Interface, 14, 24
Systems:
  Compound, 24
  Leafcell, 18

T

TAN Function, 13
Template, 14, 24
Terminal:
  Attribute, 15
  Declaration, 14
  Formal, 28

List, 14
Optional, 14
Size, 15
Vectorized, 14

V

Value Range Specification, 17
Variable:
Definition, 20
Elimination, 22
External, 20
Internal, 20
List, 20
Type, 20
Voltage Reference, 19