



Modelica® - A Unified Object-Oriented Language for Physical Systems Modeling

Language Specification

Version 3.0

September 5, 2007

Abstract

This document defines the Modelica¹ language, version 3.0, which is developed by the Modelica Association, a non-profit organization with seat in Linköping, Sweden. Modelica is a freely available, object-oriented language for modeling of large, complex, and heterogeneous physical systems. It is suited for multi-domain modeling, for example, mechatronic models in robotics, automotive and aerospace applications involving mechanical, electrical, hydraulic and control subsystems, process oriented applications and generation and distribution of electric power. Models in Modelica are mathematically described by differential, algebraic and discrete equations. No particular variable needs to be solved for manually. A Modelica tool will have enough information to decide that automatically. Modelica is designed such that available, specialized algorithms can be utilized to enable efficient handling of large models having more than one hundred thousand equations. Modelica is suited and used for hardware-in-the-loop simulations and for embedded control systems. More information is available at <http://www.Modelica.org/>

¹ Modelica is a registered trademark of the Modelica Association

Copyright © 1998-2007, Modelica Association (<http://www.Modelica.org>)

All rights reserved. Reproduction or use of editorial or pictorial content is permitted, i.e., this document can be freely distributed especially electronically, provided the copyright notice and these conditions are retained. No patent liability is assumed with respect to the use of information contained herein. While every precaution has been taken in the preparation of this document no responsibility for errors or omissions is assumed.

The contributors to this and to previous versions of this document are listed in Appendix D. All contributors worked voluntarily and without compensation.

Table of Contents

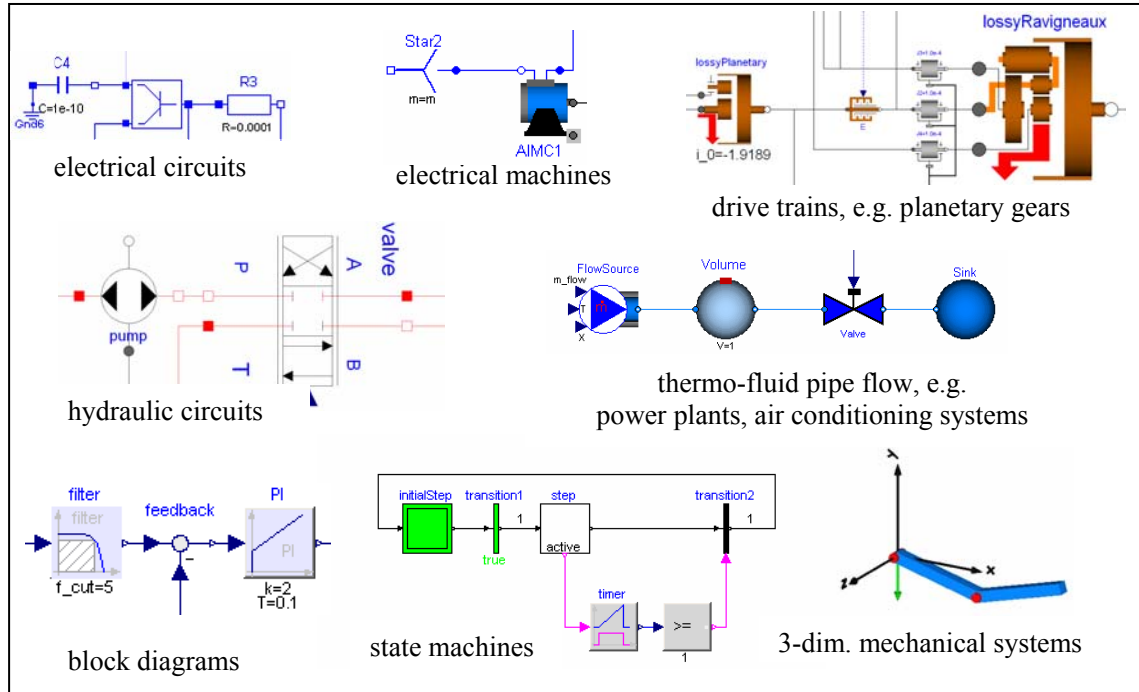
Preface	6
Chapter 1 Introduction	9
1.1 Overview of Modelica	9
1.2 Scope of the Specification	9
1.3 Some Definitions	10
1.4 Notation and Grammar	10
Chapter 2 Lexical Structure	11
2.1 Character Set	11
2.2 Comments	11
2.3 Identifiers, Names, and Keywords	12
2.4 Literal Constants	12
2.5 Operator Symbols	14
Chapter 3 Operators and Expressions	15
3.1 Expressions	15
3.2 Operator Precedence and Associativity	15
3.3 Evaluation Order	16
3.4 Arithmetic Operators	17
3.5 Equality, Relational, and Logical Operators	17
3.6 Miscellaneous Operators and Variables	18
3.7 Built-in Intrinsic Operators with Function Syntax	19
3.8 Variability of Expressions	25
Chapter 4 Classes, Predefined Types, and Declarations	29
4.1 Access Control – Public and Protected Elements	29
4.2 Double Declaration not Allowed	29
4.3 Declaration Order and Usage before Declaration	29
4.4 Component Declarations	29
4.5 Class Declarations	34
4.6 Specialized Classes	36
4.7 Balanced Models	37
4.8 Predefined Types	43
Chapter 5 Scoping, Name Lookup, and Flattening	47
5.1 Flattening Context	47
5.2 Enclosing Classes	47
5.3 Static Name Lookup	47
5.4 Instance Hierarchy Name Lookup of Inner Declarations	49
5.5 Simultaneous Inner/Outer Declarations	50
5.6 Flattening Process	51
Chapter 6 Interface or Type Relationships	53
6.1 The Concepts of Type, Interface and Subtype	54
6.2 Interface or Type	54

6.3	Interface Compatibility or Subtyping	56
6.4	Plug-Compatibility or Restricted Subtyping.....	57
6.5	Function-Compatibility or Function-Subtyping for Functions	59
6.6	Type Compatible Expressions	60
Chapter 7	Inheritance, Modification, and Redefinition.....	61
7.1	Inheritance—Extends Clause.....	61
7.2	Modifications	63
7.3	Redefinition	67
Chapter 8	Equations.....	73
8.1	Equation Categories	73
8.2	Flattening and Lookup in Equations	73
8.3	Equations in Equation Sections.....	73
8.4	Synchronous Data-flow Principle and Single Assignment Rule.....	79
8.5	Events and Synchronization.....	80
8.6	Initialization, initial equation, and initial algorithm.....	82
Chapter 9	Connectors and Connections.....	85
9.1	Connect-Equations and Connectors	85
9.2	Generation of Connection Equations	89
9.3	Restrictions of Connections and Connectors	90
9.4	Equation Operators for Overconstrained Connection-Based Equation Systems	92
Chapter 10	Arrays.....	99
10.1	Array Declarations	99
10.2	Flexible Array Sizes.....	101
10.3	Built-in Array Functions	101
10.4	Vector, Matrix and Array Constructors	104
10.5	Array Indexing.....	107
10.6	Scalar, Vector, Matrix, and Array Operator Functions.....	109
10.7	Empty Arrays.....	113
Chapter 11	Statements and Algorithm Sections.....	115
11.1	Algorithm Sections	115
11.2	Statements	115
Chapter 12	Functions	121
12.1	Function Declaration.....	121
12.2	Function as a Specialized Class	122
12.3	Pure Modelica Functions	123
12.4	Function Call.....	124
12.5	Built-in Functions	128
12.6	Record Constructor Functions	128
12.7	Declaring Derivatives of Functions	130
12.8	External Function Interface.....	133
Chapter 13	Packages	143
13.1	Package as Specialized Class.....	143

13.2	Motivation and Usage of Packages	143
Chapter 14	Annotations	147
14.1	Vendor-Specific Annotations	147
14.2	Annotations for Documentation	147
14.3	Annotations for Code Generation	148
14.4	Annotations for Simulation Experiments	149
14.5	Annotations for Graphical Objects	149
14.6	Annotations for the Graphical User Interface	157
14.7	Annotations for Version Handling	158
14.8	Annotations for Functions	159
14.9	Annotation Choices for Modifications and Redclarations	159
Chapter 15	Unit Expressions	161
15.1	The Syntax of Unit Expressions	161
15.2	Examples	162
Chapter 16	The Modelica Standard Library	163
Appendix A	Glossary	167
Appendix B	Modelica Concrete Syntax	171
Appendix C	Modelica DAE Representation	178
Appendix D	Modelica Revision History	180
Index	191

Preface

Modelica is a freely available, object-oriented language for modeling of large, complex, and heterogeneous physical systems. From a user's point of view, models are described by schematics, also called object diagrams. Examples are shown in the next figure:



A schematic consists of connected components, like a resistor, or a hydraulic cylinder. A component has “connectors” (often also called “ports”) that describe the interaction possibilities, e.g., an electrical pin, a mechanical flange, or an input signal. By drawing connection lines between connectors a physical system or block diagram model is constructed. Internally a component is defined by another schematic or on “bottom” level, by an equation based description of the model in Modelica syntax.

The Modelica language is a textual description to define all parts of a model and to structure model components in libraries, called packages. An appropriate Modelica simulation environment is needed to graphically edit and browse a Modelica model (by interpreting the information defining a Modelica model) and to perform model simulations and other analysis. Information about such environments is available at www.modelica.org/tools. Basically, all Modelica language elements are mapped to differential, algebraic and discrete equations. There are no language elements to describe directly partial differential equations, although some types of discretized partial differential equations can be reasonably defined, e.g., based on the finite volume method and there are Modelica libraries to import results of finite-element programs.

This document defines the details of the Modelica language. It is not intended to learn the Modelica language with this text. There are better alternatives, such as the Modelica books referenced at www.modelica.org/publications. This specification is used by computer scientist to implement a Modelica translator and by modelers who want to understand the exact details of a particular language element.

The Modelica language has been developed since 1996. This is the third main release. With the previous Modelica release (2.2) the practical limit for Modelica models is around several 100000 equations. Modelica 3.0 introduces nearly no new features, but redesigns the “kernel” by the introduction of mild restrictions. The main purpose is to provide a “cleaned-up” version so that the practical limit on the model size is *increased by an order of magnitude*. All previous *eight* Modelica releases have been backwards compatible. Modelica 3.0 is the first version that is slightly non-backwards compatible to the previous version, but an automated conversion is possible. The main new features of Modelica 3.0 are:

- The new concept of “balanced models” is introduced, which basically means that on every level the “number of equations” and the “number of unknowns” must be identical. As a result, when components pass a Modelica translator successfully, it is guaranteed that the system model consisting of such components has always the property that the “number of equations” of the global model is identical to the “number of all unknowns”. In previous versions this was not the case and modelers had often a hard time to figure out the source of a modeling error, if the global count of equations and unknowns did not match. The feature of balanced models also allows compiling model components separately.
- The type system, or equivalently the concept of interface of components and classes, has been redesigned and defined more precisely. As a result, if a replaceable component is redeclared (i.e., a component from a class A is replaced by a component of a class B), it is guaranteed that the redeclared component again leads to a balanced model and that no language error can occur by this redeclaration. Previously, a redeclaration could lead to a wrong model and it could be difficult to figure out the source of the error. It is, however, still possible, e.g., that the simulation of the redeclared model fails, because the model becomes numerically singular due to the redeclaration (e.g. due to too idealized model).
- The graphical annotations that define the graphical layout of components and of component connections (= schematic) have been redesigned, based on practical experience with the Modelica 2 graphical annotations. Also, new graphical annotations have been introduced, e.g. to have Bezier-Splines as line style and annotations for schematic animations and for interactive user input.
- The Modelica specification text was largely re-structured, completely new written and improved in many ways, e.g., more meaningful examples, fixing flaws in the language, and defining language elements more precisely.

Sept. 5, 2007, The Modelica Association

Chapter 1

Introduction

1.1 Overview of Modelica

Modelica is a language for modeling of physical systems, designed to support effective library development and model exchange. It is a modern language built on acausal modeling with mathematical equations and object-oriented constructs to facilitate reuse of modeling knowledge.

1.2 Scope of the Specification

The semantics of the Modelica language is specified by means of a set of rules for translating any class described in the Modelica language to a flat Modelica structure. A class must have additional properties in order that its flat Modelica structure can be further transformed into a set of differential, algebraic and discrete equations (= flat hybrid DAE). Such classes are called simulation models.

The flat Modelica structure is also defined for other cases than simulation models; including functions (can be used to provide algorithmic contents), packages (used as a structuring mechanism), and partial models (used as base-models). This allows correctness to be verified before building the simulation model.

Modelica was designed to facilitate symbolic transformations of models, especially by mapping basically every Modelica language construct to continuous or instantaneous equations in the flat Modelica structure. Many Modelica models, especially in the associated Modelica Standard Library, are higher index systems, and can only be reasonably simulated if symbolic index reduction is performed, i.e., equations are differentiated and appropriate variables are selected as states, so that the resulting system of equations can be transformed to state space form (at least locally numerically), i.e., a hybrid DAE of index zero. The Modelica specification does not define how to simulate a model. However, it defines a set of equations that the simulation result should satisfy as well as possible.

The key issues of the translation (or flattening) are:

- Expansion of inherited base classes
- Parameterization of base classes, local classes and components
- Generation of connection equations from connect-equations

The flat hybrid DAE form consists of:

- Declarations of variables with the appropriate basic types, prefixes and attributes, such as "parameter Real v=5".
- Equations from equation sections.
- Function invocations where an invocation is treated as a set of equations which involves all input and all result variables (number of equations = number of basic result variables).
- Algorithm sections where every section is treated as a set of equations which involves the variables occurring in the algorithm section (number of equations = number of different assigned variables).
- When-clauses where every when-clause is treated as a set of conditionally evaluated equations, also called instantaneous equations, which are functions of the variables occurring in the clause (number of equations = number of different assigned variables).

Therefore, a flat hybrid DAE is seen as a set of equations where some of the equations are only conditionally evaluated (e.g. instantaneous equations are only evaluated when the corresponding when-condition becomes true). Initial setup of the model is specified using start-values and instantaneous equations that hold at the initial time only.

A Modelica class may also contain annotations, i.e. formal comments, which specify graphical representations of the class (icon and diagram), documentation text for the class, and version information.

1.3 Some Definitions

The semantic specification should be read together with the Modelica grammar. Non-normative text, i.e., examples and comments, are enclosed in []; comments are set in italics. Additional terms are explained in the glossary in Appendix A. Some important terms are:

<i>Term</i>	<i>Definition</i>
Component	An element defined by the production <code>component_clause</code> in the Modelica grammar (basically a variable or an instance of a class)
Element	Class definitions, extends-clauses and component-clauses declared in a class (basically a class reference or a component in a declaration).
Flattening	The translation of a model described in Modelica to the corresponding model described as a hybrid DAE, involving expansion of inherited base classes, parameterization of base classes, local classes and components, and generation of connection equations from connect-equations (basically, mapping the hierarchical structure of a model into a set of differential, algebraic and discrete equations together with the corresponding variable declarations and function definitions from the model).

1.4 Notation and Grammar

The following syntactic meta symbols are used (extended BNF):

```
[ ] optional
{ } repeat zero or more times
```

Boldface denotes keywords of the Modelica language. Keywords are reserved words and may not be used as identifiers, with the exception of `initial` which is a keyword in section headings, but it is also possible to call the function `initial()`.

See Appendix B for a full lexical specification and grammar.

Chapter 2

Lexical Structure

This chapter describes several of the basic building blocks of Modelica such as characters and lexical units including identifiers and literals. Without question, the smallest building blocks in Modelica are single characters belonging to a character set. Characters are combined to form lexical units, also called tokens. These tokens are detected by the lexical analysis part of the Modelica translator. Examples of tokens are literal constants, identifiers, and operators. Comments are not really lexical units since they are eventually discarded. On the other hand, comments are detected by the lexical analyzer before being thrown away.

The information presented here is derived from the more formal specification in Appendix B.

2.1 Character Set

The character set of the Modelica language is not yet completely specified. However, in practice the currently available Modelica tools work well for code written in the 8-bit Latin-1 character set, which corresponds to the first 256 characters of the Unicode character set. Most of the first 128 characters of Latin-1 are equivalent to the 7-bit ASCII character set.

2.2 Comments

There are three kinds of comments in Modelica which are not lexical units in the language and therefore are ignored by a Modelica translator. [*The comment syntax is identical to that of C++*]. The following comment variants are available:

<code>// comment</code>	Characters from <code>//</code> to the end of the line are ignored.
<code>/* comment */</code>	Characters between <code>/*</code> and <code>*/</code> are ignored, including line terminators.

Modelica comments do not nest, i.e., `/* */` cannot be embedded within `/* */`. The following is *invalid*:

```
/* Commented out - erroneous comment, invalid nesting of comments!
/* This is a interesting model */
model interesting
...
end interesting;
*/
```

There is also a kind of “documentation comment,” really a *documentation string* that is part of the Modelica language and therefore not ignored by the Modelica translator. Such “comments” may occur at the ends of declarations, equations, or statements or at the beginning of class definitions. For example:

```
model TempResistor "Temperature dependent resistor"
...
parameter Real R "Resistance for reference temp.";
...
```

```
end TempResistor;
```

2.3 Identifiers, Names, and Keywords

Identifiers are sequences of letters, digits, and other characters such as underscore, which are used for *naming* various items in the language. Certain combinations of letters are *keywords* represented as *reserved* words in the Modelica grammar and are therefore not available as identifiers.

2.3.1 Identifiers

Modelica *identifiers*, used for naming classes, variables, constants, and other items, are of two forms. The first form always start with a letter or underscore (`_`), followed by any number of letters, digits, or underscores. Case is significant, i.e., the names `Inductor` and `inductor` are different. The second form (`Q-IDENT`) starts with a single quote, followed by a sequence of any characters, where single-quote must be preceded by backslash, and terminated by a single quote, e.g. `'12H'`, `'13\'H'`, `'+foo'`. The following BNF-like rules define Modelica identifiers, where curly brackets `{}` indicate repetition zero or more times, and vertical bar `|` indicates alternatives. A full BNF definition of the Modelica syntax and lexical units is available in Appendix B.

```
IDENT      = NONDIGIT { DIGIT | NONDIGIT } | Q-IDENT
Q-IDENT    = "'" { Q-CHAR | S-ESCAPE } "'"
NONDIGIT   = "_" | letters "a" to "z" | letters "A" to "Z"
DIGIT      = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Q-CHAR     = any member of the source character set except single-quote "'", and backslash "\"
S-ESCAPE   = "\"'\" | "\"\"\" | "\"?\" | "\"\\\" |
              "\"a\" | "\"b\" | "\"f\" | "\"n\" | "\"r\" | "\"t\" | "\"v\""
```

2.3.2 Names

A *name* is an identifier with a certain interpretation or meaning. For example, a name may denote an `Integer` variable, a `Real` variable, a function, a type, etc. A name may have different meanings in different parts of the code, i.e., different scopes. The interpretation of identifiers as names is described in more detail in Chapter 5. The meaning of package names is described in more detail in Chapter 13.

2.3.3 Modelica Keywords

The following Modelica *keywords* are reserved words and may not be used as identifiers:

algorithm	and	annotation	assert	block
break	class	connect	connector	constant
constrainedby	der	discrete	else	elseif
elsewhen	encapsulated	end	enumeration	equation
expandable	extends	external	false	final
flow	for	function	if	in
initial	inner	input	loop	model
not	or	outer	output	package
parameter	partial	protected	public	record
redeclare	replaceable	return	then	true
type	when	while	within	

2.4 Literal Constants

Literal constants are unnamed constants that have different forms depending on their type. Each of the predefined types in Modelica has a way of expressing unnamed constants of the corresponding type, which is presented in the ensuing subsections. Additionally, array literals and record literals can be expressed.

2.4.1 Floating Point Numbers

A floating point number is expressed as a decimal number in the form of a sequence of decimal digits optionally followed by a decimal point, optionally followed by an exponent. At least one digit must be present. The exponent is indicated by an E or e, followed by an optional sign (+ or −) and one or more decimal digits. The minimal recommended range is that of IEEE double precision floating point numbers, for which the largest representable positive number is 1.7976931348623157E+308 and the smallest positive number is 2.2250738585072014E−308. For example, the following are floating point number literal constants:

```
22.5,  3.141592653589793, 1.2E-35
```

The same floating point number can be represented by different literals. For example, all of the following literals denote the same number:

```
13.,  13E0,  1.3e1,  .13E2
```

2.4.2 Integer Literals

Literals of type `Integer` are sequences of decimal digits, e.g. as in the integer numbers 33, 0, 100, 30030044. [Negative numbers are formed by unary minus followed by an integer literal]. The minimal recommended number range is from −2147483648 to +2147483647 for a two's-complement 32-bit integer implementation.

2.4.3 Boolean Literals

The two Boolean literal values are `true` and `false`.

2.4.4 Strings

String literals appear between double quotes as in "between". Any character in the Modelica language character set apart from double quote (") and backslash (\), including new-line, can be *directly* included in a string without using an escape code. Certain characters in string literals can be represented using escape codes, i.e., the character is preceded by a backslash (\) within the string. Those characters are:

\'	single quote—may also appear without backslash in string constants.
\"	double quote
\?	question-mark—may also appear without backslash in string constants.
\\	backslash itself
\a	alert (bell, code 7, ctrl-G)
\b	backspace (code 8, ctrl-H)
\f	form feed (code 12, ctrl-L)
\n	new-line (code 10, ctrl-J)
\r	return (code 13, ctrl-M)
\t	horizontal tab (code 9, ctrl-I)
\v	vertical tab (code 11, ctrl-K)

For example, a string literal containing a tab, the words: `This is`, double quote, space, the word: `between`, double quote, space, the word: `us`, and new-line, would appear as follows:

```
"\tThis is\" between\" us\n"
```

Concatenation of string literals in certain situations (see the Modelica grammar) is denoted by the + operator in Modelica, e.g. "a" + "b" becomes "ab". This is useful for expressing long string literals that need to be written on several lines.

[Note, if the contents of a file is read into a Modelica string, it is assumed that the reading function is responsible to handle the different line ending symbols on file (e.g. on Linux systems to have a “newline” character at the end of a line and on Windows systems to have a “newline” and a “carriage return” character. As usual in programming languages, the content of a file in a Modelica string only contains the “newline” character.

For long string comments, e.g., the “info” annotation to store the documentation of a model, it would be very inconvenient, if the string concatenation operator would have to be used for every line of documentation. It is assumed that a Modelica tool supports the non-printable “newline” character when browsing or editing a string literal. For example, the following statement defines one string that contains (non-printable) newline characters:

```
    assert(noEvent(length > s_small), "  
The distance between the origin of frame_a and the origin of frame_b  
of a LineForceWithMass component became smaller as parameter s_small  
(= a small number, defined in the \"Advanced\" menu). The distance is  
set to s_small, although it is smaller, to avoid a division by zero  
when computing the direction of the line force.",  
    level = AssertionLevel.warning);  
]
```

2.5 Operator Symbols

The predefined operator symbols are formally defined on page 171 and summarized in the table of operators in Section 3.2.

Chapter 3

Operators and Expressions

The lexical units are combined to form even larger building blocks such as expressions according to the rules given by the expression part of the Modelica grammar in Appendix B.

This chapter describes the evaluation rules for expressions, the concept of expression variability, built-in mathematical operators and functions, and the built-in special Modelica operators with function syntax.

Expressions can contain variables and constants, which have types, predefined or user defined. The predefined built-in types of Modelica are `Real`, `Integer`, `Boolean`, `String`, and enumeration types which are presented in more detail in Section 4.7. [*The abbreviated predefined type information below is given as background information for the rest of the presentation.*]

3.1 Expressions

Modelica equations, assignments and declaration equations contain expressions.

Expressions can contain basic operations, `+`, `-`, `*`, `/`, `^`, etc. with normal precedence as defined in the Table in Section 3.2 and the grammar in Appendix B. The semantics of the operations is defined for both scalar and array arguments in Section 10.6.

It is also possible to define functions and call them in a normal fashion. The function call syntax for both positional and named arguments is described in Section 12.4.1 and for vectorized calls in Section 12.4.3. The built-in array functions are given in Section 10.1.1 and other built-in operators in Section 3.7.

3.2 Operator Precedence and Associativity

Operator precedence determines the order of evaluation of operators in an expression. An operator with higher precedence is evaluated before an operator with lower precedence in the same expression.

The following table presents all the expression operators in order of precedence from highest to lowest, as derived from the Modelica grammar in Appendix B. All operators are binary except exponentiation, the postfix operators and those shown as unary together with *expr*, the conditional operator, the array construction operator `{}` and concatenation operator `[]`, and the array range constructor which is either binary or ternary. Operators with the same precedence occur at the same line of the table:

Table 3-1. Operators.

Operator Group	Operator Syntax	Examples
postfix array index operator	<code>[]</code>	<code>arr[index]</code>
postfix access operator	<code>.</code>	<code>a.b</code>
postfix function call	<code>funcName(function-arguments)</code>	<code>sin(4.36)</code>
array construct/concat	<code>{expressions}</code> <code>[expressions]</code>	<code>{2,3}</code> <code>[5,6]</code>
	<code>[expressions; expressions...]</code>	<code>[2,3; 7,8]</code>
exponentiation	<code>^</code>	<code>2^3</code>

multiplicative and array elementwise multiplicative	<code>*</code> <code>/</code> <code>.*</code> <code>./</code>	<code>2*3</code> <code>2/3</code> <code>[1,2;3,4].*[2,3;5,6]</code>
additive and array elementwise additive	<code>+</code> <code>-</code> <code>+expr</code> <code>-expr</code> <code>.+</code> <code>.-</code>	<code>a+b</code> , <code>a-b</code> , <code>+a</code> , <code>-a</code> <code>[1,2;3,4].+[2,3;5,6]</code>
relational	<code><</code> <code><=</code> <code>></code> <code>>=</code> <code>==</code> <code><></code>	<code>a<b</code> , <code>a<=b</code> , <code>a>b</code> , ...
...		
unary negation	<code>not expr</code>	<code>not b1</code>
logical and	<code>and</code>	<code>b1 and b2</code>
logical or	<code>or</code>	<code>b1 or b2</code>
array range	<code>expr : expr : expr</code>	<code>1:5</code> , <code>start:step:stop</code>
conditional	<code>if expr then expr else expr</code>	<code>if b then 3 else x</code>
named argument	<code>ident = expr</code>	<code>x = 2.26</code>

The conditional operator may also include elseif-clauses. Equality `=` and assignment `:=` are not expression operators since they are allowed only in equations and in assignment statements respectively. All binary expression operators are left associative.

[The unary minus and plus in Modelica is slightly different than in Mathematica and in MATLAB², since the following expressions are illegal (whereas in Mathematica³ and in MATLAB these are valid expressions):

```
2*-2 // = -4 in Mathematica/MATLAB; is illegal in Modelica
--2  // =  2 in Mathematica/MATLAB; is illegal in Modelica
++2  // =  2 in Mathematica/MATLAB; is illegal in Modelica
2--2 // =  4 in Mathematica/MATLAB; is illegal in Modelica
```

]

3.3 Evaluation Order

A tool is free to solve equations, reorder expressions and to not evaluate expressions if their values do not influence the result (e.g. short-circuit evaluation of Boolean expressions). If-statements and if-expressions guarantee that their clauses are only evaluated if the appropriate condition is true, but relational operators generating state or time events will during continuous integration have the value from the most recent event.

If a numeric operation overflows the result is undefined. For literals it is recommend to automatically convert the number to another type with greater precision.

3.3.1 Example: Guarding Expressions Against Incorrect Evaluation

[Example. If one wants to guard an expression against incorrect evaluation, it should be guarded by an if:

```
Boolean v[n];
Boolean b;
Integer I;
equation
  x=v[I] and (I>=1 and I<=n); // Invalid
  x=if (I>=1 and I<=n) then v[I] else false; // Correct
```

To guard square against square root of negative number use noEvent:

```
der(h)=if h>0 then -c*sqrt(h) else 0; // Incorrect
der(h)=if noEvent(h>0) then -c*sqrt(h) else 0; // Correct
```

]

² MATLAB is a registered trademark of MathWorks Inc.

³ Mathematica is a registered trademark of Wolfram Research Inc.

3.4 Arithmetic Operators

Modelica supports five binary arithmetic operators that operate on any numerical type:

\wedge	Exponentiation
$*$	Multiplication
$/$	Division
$+$	Addition
$-$	Subtraction

Some of these operators can also be applied to a combination of a scalar type and an array type, see Section 10.6.

The syntax of these operators is defined by the following rules from the Modelica grammar:

```

arithmetic_expression :
  [ add_op ] term { add_op term }

add_op :
  "+" | "-"

term :
  factor { mul_op factor }

mul_op :
  "*" | "/"

factor :
  primary [ "^" primary ]

```

3.5 Equality, Relational, and Logical Operators

Modelica supports the standard set of relational and logical operators, all of which produce the standard boolean values `true` or `false`.

$>$	greater than
$>=$	greater than or equal
$<$	less than
$<=$	less than or equal to
$==$	equality within expressions
$<>$	Inequality

A single equals sign `=` is never used in relational expressions, only in equations (Chapter 8, Section 10.6.1) and in function calls using named parameter passing (Section 12.4.1).

The following logical operators are defined:

<code>not</code>	negation, unary operator
<code>and</code>	logical and
<code>or</code>	logical or

The grammar rules define the syntax of the relational and logical operators.

```

logical_expression :
  logical_term { or logical_term }

logical_term :
  logical_factor { and logical_factor }

logical_factor :
  [ not ] relation

```

```
relation :  
  arithmetic_expression [ rel_op arithmetic_expression ]
```

```
rel_op :  
  "<" | "<=" | ">" | ">=" | "==" | "<>"
```

The following holds for relational operators:

- Relational operators $<$, $<=$, $>$, $>=$, $==$, $<>$, are only defined for scalar operands of simple types. The result is Boolean and is true or false if the relation is fulfilled or not, respectively.
- For operands of type `String`, `str1 op str2` is for each relational operator, `op`, defined in terms of the C-function `strcmp` as `strcmp(str1, str2) op 0`.
- For operands of type `Boolean`, `false < true`.
- For operands of enumeration types, the order is given by the order of declaration of the enumeration literals.
- In relations of the form `v1 == v2` or `v1 <> v2`, `v1` or `v2` shall, unless used in a function, not be a subtype of `Real`. [The reason for this rule is that relations with `Real` arguments are transformed to state events (see Events, Section 8.5) and this transformation becomes unnecessarily complicated for the `==` and `<>` relational operators (e.g. two crossing functions instead of one crossing function needed, epsilon strategy needed even at event instants). Furthermore, testing on equality of `Real` variables is questionable on machines where the number length in registers is different to number length in main memory].
- Relations of the form “`v1 rel_op v2`”, with `v1` and `v2` variables and `rel_op` a relational operator are called elementary relations. If either `v1` or `v2` or both variables are a subtype of `Real`, the relation is called a `Real` elementary relation.

3.6 Miscellaneous Operators and Variables

Modelica also contains a few built-in operators which are not standard arithmetic, relational, or logical operators. These are described below, including `time`, which is a built-in variable, not an operator.

3.6.1 String Concatenation

Concatenation of strings (see the Modelica grammar) is denoted by the `+` operator in Modelica [e.g. `"a" + "b"` becomes `"ab"`].

3.6.2 Array Constructor Operator

The array constructor operator `{ ... }` is described in Section 10.4.

3.6.3 Array Concatenation Operator

The array concatenation operator `[...]` is described in Section 10.4.2.

3.6.4 Array Range Operator

The array range constructor operator `:` is described in Section 10.4.3.

3.6.5 If-Expressions

An expression

```
if expression1 then expression2 else expression3
```

is one example of if-expression. First `expression1`, which must be boolean expression, is evaluated. If `expression1` is true `expression2` is evaluated and is the value of the if-expression, else `expression3` is evaluated and is the value of the if-expression. The two expressions, `expression2` and `expression3`, must be type compatible expressions (Section 6.6) giving the type of the if-expression. If-expressions with `elseif` are defined by replacing `elseif` by `else if`. [Note: `elseif` has been added for symmetry with if-clauses.] For short-circuit evaluation see Section 3.3.

[Example:

```
Integer i;
Integer sign_of_i1=if i<0 then -1 elseif i==0 then 0 else 1;
Integer sign_of_i2=if i<0 then -1 else if i==0 then 0 else 1;
]
```

3.6.6 Member Access Operator

It is possible to access members of a class instance using dot notation, i.e., the `.` operator.

[Example: `R1.R` for accessing the resistance component `R` of resistor `R1`. Another use of dot notation: local types and inner classes which are members of a class can of course also be accessed using dot notation on the name of the class, not on instances of the class.]

3.6.7 Built-in Variable time

All declared variables are functions of the independent variable `time`. The variable `time` is a built-in variable available in all classes, which is treated as an input variable. It is implicitly defined as:

```
input Real time (final quantity = "Time",
                final unit      = "s");
```

The value of the start attribute of `time` is set to the time instant at which the simulation is started.

[Example:

```
encapsulated model SineSource
  import Modelica.Math.sin;
  connector OutPort=output Real;
  OutPort y=sin(time); // Uses the built-in variable time.
end SineSource;
]
```

3.7 Built-in Intrinsic Operators with Function Syntax

Certain built-in operators of Modelica have the same syntax as a function call. However, they do not behave as a mathematical function, because the result depends not only on the input arguments but also on the status of the simulation.

There are also built-in functions that depend only on the input argument, but also may trigger events in addition to returning a value. Intrinsic means that they are defined at the Modelica language level, not in the Modelica library. The following built-in intrinsic operators/functions are available:

- Mathematical functions and conversion functions, see Section 3.7.1 below.
- Derivative and special purpose operators with function syntax, see Section 3.7.2 below.
- Event-related operators with function syntax, see Section 3.7.3 below.
- Array operators/functions, see Section 10.1.1.

With exception of built-in operator `String(..)`, all operators in this section can only be called with positional arguments.

3.7.1 Numeric Functions and Conversion Functions

The following mathematical operators and functions, also including some conversion functions, are predefined in Modelica, and are vectorizable according to Section 12.4.5, except for the `String` function. The functions which do not trigger events are described in the table below, whereas the event-triggering mathematical functions are described in Section 3.7.1.1.

<code>abs(v)</code>	Is expanded into “ <code>noEvent(if v >= 0 then v else -v)</code> ”. Argument <code>v</code> needs to be an Integer or Real expression.
<code>sign(v)</code>	Is expanded into “ <code>noEvent(if v>0 then 1 else if v<0 then -1 else 0)</code> ”. Argument <code>v</code> needs to be an Integer or Real expression.
<code>sqrt(v)</code>	Returns the square root of <code>v</code> if <code>v</code> ≥0, otherwise an error occurs. Argument <code>v</code> needs to be an Integer or Real expression.
<code>Integer(e)</code>	Returns the ordinal number of the enumeration value <code>E.enumvalue</code> , where <code>Integer(E.e1)=1</code> , <code>Integer(E.en)=size(E)</code> , for an enumeration type <code>E=enumeration(e1, ..., en)</code> . See also Section 4.8.5.2.
<code>String(b, <options>)</code> <code>String(i, <options>)</code> <code>String(r, significantDigits=d, <options>)</code> <code>String(r, format=s)</code> <code>String(e, <options>)</code>	<p>Convert a scalar non-String expression to a String representation. The first argument may be a Boolean <code>b</code>, an Integer <code>i</code>, a Real <code>r</code> or an Enumeration <code>e</code> (Section 4.8.5.2). The optional <code><options></code> are:</p> <p>Integer <code>minimumLength=0</code>: minimum length of the resulting string. If necessary, the blank character is used to fill up unused space.</p> <p>Boolean <code>leftJustified = true</code>: if true, the converted result is left justified in the string; if false it is right justified in the string.</p> <p>For Real expressions the output shall be according to the Modelica grammar. Integer <code>significantDigits=6</code>: defines the number of significant digits in the result string. [Examples: “12.3456”, “0.0123456”, “12345600”, “1.23456E-10”]. The format string corresponding to options is:</p> <ul style="list-style-type: none"> • for Reals: (if <code>leftJustified</code> then “-” else “”)+<code>String(minimumLength)</code>+“.”+<code>String(significantDigits)</code>+“g”, • for Integers: (if <code>leftJustified</code> then “-” else “”)+<code>String(minimumLength)</code>+“d”. <p>Format string: According to ANSI-C the format string specifies one conversion specifier (excluding the leading %), may not contain length modifiers, and may not use “*” for width and/or precision. For all numeric values the format specifiers <code>f</code>, <code>e</code>, <code>E</code>, <code>g</code>, <code>G</code> are allowed. For integral values it is also allowed to use the <code>d</code>, <code>i</code>, <code>o</code>, <code>x</code>, <code>X</code>, <code>u</code>, and <code>c</code>-format specifiers (for non-integral values a tool may round, truncate or use a different format if the integer conversion characters are used).</p> <p>The <code>x</code>,<code>X</code>-formats (hexa-decimal) and <code>c</code> (character) for Integers does not lead to input that agrees with the Modelica-grammar.</p>

3.7.1.1 Event Triggering Mathematical Functions

The built-in operators in this section trigger state events if used outside of a when-clause. [*If this is not desired, the noEvent function can be applied to them. E.g. noEvent (integer(v))*]

div(x,y)	Returns the algebraic quotient x/y with any fractional part discarded (also known as truncation toward zero). [Note: this is defined for / in C99; in C89 the result for negative numbers is implementation-defined, so the standard function div() must be used.]. Result and arguments shall have type Real or Integer. If either of the arguments is Real the result is Real otherwise Integer.
mod(x,y)	Returns the integer modulus of x/y , i.e. $\text{mod}(x,y) = x - \text{floor}(x/y) * y$. Result and arguments shall have type Real or Integer. If either of the arguments is Real the result is Real otherwise Integer. [Note, outside of a when-clause state events are triggered when the return value changes discontinuously. Examples $\text{mod}(3, 1.4) = 0.2$, $\text{mod}(-3, 1.4) = 1.2$, $\text{mod}(3, -1.4) = -1.2$]
rem(x,y)	Returns the integer remainder of x/y , such that $\text{div}(x,y) * y + \text{rem}(x, y) = x$. Result and arguments shall have type Real or Integer. If either of the arguments is Real the result is Real otherwise Integer. [Note, outside of a when-clause state events are triggered when the return value changes discontinuously. Examples $\text{rem}(3, 1.4) = 0.2$, $\text{rem}(-3, 1.4) = -0.2$]
ceil(x)	Returns the smallest integer not less than x. Result and argument shall have type Real. [Note, outside of a when-clause state events are triggered when the return value changes discontinuously.]
floor(x)	Returns the largest integer not greater than x. Result and argument shall have type Real. [Note, outside of a when-clause state events are triggered when the return value changes discontinuously.]
integer(x)	Returns the largest integer not greater than x. The argument shall have type Real. The result has type Integer. [Note, outside of a when-clause state events are triggered when the return value changes discontinuously.]

3.7.1.2 Built-in Mathematical Functions and External Built-in Functions

The following built-in mathematical functions are available in Modelica and can be called directly without any package prefix added to the function name. They are also available as external built-in functions in the Modelica.Math library.

sin(x)	sine
cos(x)	cosine
tan(x)	tangent (x shall not be: ..., $-\pi/2$, $\pi/2$, $3\pi/2$, ...)
asin(x)	inverse sine ($-1 \leq x \leq 1$)
acos(x)	inverse cosine ($-1 \leq x \leq 1$)
atan(x)	inverse tangent
atan2(x,y)	four quadrant inverse tangent
sinh(x)	hyperbolic sine
cosh(x)	hyperbolic cosine
tanh(x)	hyperbolic tangent

<code>exp(x)</code>	exponential, base e
<code>log(x)</code>	natural (base e) logarithm ($x > 0$)
<code>log10(x)</code>	base 10 logarithm ($x > 0$)

3.7.2 Derivative and Special Purpose Operators with Function Syntax

The following derivative operator and special purpose operators with function syntax are predefined:

<code>der(expr)</code>	The time derivative of <code>expr</code> . If the expression <code>expr</code> is a scalar it needs to be a subtype of Real. The expression and all its subexpressions must be differentiable. If <code>expr</code> is an array, the operator is applied to all elements of the array. For non-scalar arguments the function is vectorized according to Section 10.6.12. <i>[For Real parameters and constants the result is a zero scalar or array of the same size as the variable.]</i>
<code>delay(expr, delayTime, delayMax)</code> <code>delay(expr, delayTime)</code>	Returns: <code>expr(time-delayTime)</code> for <code>time > time.start + delayTime</code> and <code>expr(time.start)</code> for <code>time <= time.start + delayTime</code> . The arguments, i.e., <code>expr</code> , <code>delayTime</code> and <code>delayMax</code> , need to be subtypes of Real. <code>delayMax</code> needs to be additionally a parameter expression. The following relation shall hold: <code>0 <= delayTime <= delayMax</code> , otherwise an error occurs. If <code>delayMax</code> is not supplied in the argument list, <code>delayTime</code> need to be a parameter expression. See also Section 3.7.2.1. For non-scalar arguments the function is vectorized according to Section 10.6.12.
<code>cardinality(c)</code>	<i>[This is a deprecated operator. It should no longer be used, since it will be removed in one of the next Modelica releases.]</i> Returns the number of (inside and outside) occurrences of connector instance <code>c</code> in a connect-equation as an Integer number. See also Section 3.7.2.2.
<code>semiLinear(x, positiveSlope, negativeSlope)</code>	Returns: <code>if x >= 0 then positiveSlope*x else negativeSlope*x</code> . The result is of type Real. See Section 3.7.2.3 <i>[especially in the case when $x = 0$]</i> . For non-scalar arguments the function is vectorized according to Section 10.6.12.

A few of these operators are described in more detail in the following.

3.7.2.1 delay

[The `delay()` operator allows a numerical sound implementation by interpolating in the (internal) integrator polynomials, as well as a more simple realization by interpolating linearly in a buffer containing past values of expression `expr`. Without further information, the complete time history of the delayed signals needs to be stored, because the delay time may change during simulation. To avoid excessive storage requirements and to enhance efficiency, the maximum allowed delay time has to be given via `delayMax`.

This gives an upper bound on the values of the delayed signals which have to be stored. For real-time simulation where fixed step size integrators are used, this information is sufficient to allocate the necessary storage for the internal buffer before the simulation starts. For variable step size integrators, the buffer size is dynamic during integration. In principle, a `delay` operator could break algebraic loops. For simplicity, this is not supported because the minimum delay time has to be give as additional argument to be fixed at compile time. Furthermore, the maximum step size of the integrator is limited by this minimum delay time in order to avoid extrapolation in the delay buffer.]

3.7.2.2 cardinality (deprecated)

[The cardinality operator is deprecated for the following reasons and will be removed in a future release:

- Reflective operator may make early type checking more difficult.
- Almost always abused in strange ways
- Not used for Bond graphs even though it was originally introduced for that purpose.

]

[The cardinality() operator allows the definition of connection dependent equations in a model, for example:

```
connector Pin
  Real v;
  flow Real i;
end Pin;

model Resistor
  Pin p, n;
equation
  assert(cardinality(p) > 0 and cardinality(n) > 0,
    "Connectors p and n of Resistor must be connected");
  // Equations of resistor
  ...
end Resistor;
```

]

3.7.2.3 semiLinear

(See definition of semiLinear in Section 3.7.2). In some situations, equations with the semiLinear() function become underdetermined if the first argument (x) becomes zero, i.e., there are an infinite number of solutions. It is recommended that the following rules are used to transform the equations during the translation phase in order to select one meaningful solution in such cases:

Rule 1: The equations

```
y = semiLinear(x, sa, s1);
y = semiLinear(x, s1, s2);
y = semiLinear(x, s2, s3);
...
y = semiLinear(x, sN, sb);
...
```

may be replaced by

```
s1 = if x >= 0 then sa else sb
s2 = s1;
s3 = s2;
...
sN = sN-1;
y = semiLinear(x, sa, sb);
```

Rule 2: The equations

```
x = 0;
y = 0;
y = semiLinear(x, sa, sb);
```

may be replaced by

```
x = 0
y = 0;
sa = sb;
```

[For symbolic transformations, the following property is useful (this follows from the definition):

```
semiLinear(m_flow, port_h, h);
```

is identical to :

```
-semiLinear(-m_flow, h, port_h);
```

The `semiLinear` function is designed to handle reversing flow in fluid systems, such as

```
H_flow =semiLinear(m_flow, port.h, h);
```

i.e., the enthalpy flow rate `H_flow` is computed from the mass flow rate `m_flow` and the upstream specific enthalpy depending on the flow direction.

]

3.7.3 Event-Related Operators with Function Syntax

The following event-related operators with function syntax are supported. The operators `noEvent`, `pre`, `edge`, and `change`, are vectorizable according to Section 12.4.5

<code>initial()</code>	Returns <code>true</code> during the initialization phase and <code>false</code> otherwise [<i>thereby triggering a time event at the beginning of a simulation</i>].
<code>terminal()</code>	Returns <code>true</code> at the end of a successful analysis [<i>thereby ensuring an event at the end of successful simulation</i>].
<code>noEvent(expr)</code>	Real elementary relations within <code>expr</code> are taken literally, i.e., no state or time event is triggered. See also Section 3.7.3.2 and Section 8.5.
<code>smooth(p, expr)</code>	If $p \geq 0$ <code>smooth(p, expr)</code> returns <code>expr</code> and states that <code>expr</code> is p times continuously differentiable, i.e.: <code>expr</code> is continuous in all real variables appearing in the expression and all partial derivatives with respect to all appearing real variables exist and are continuous up to order p . The only allowed types for <code>expr</code> in <code>smooth</code> are: real expressions, arrays of allowed expressions, and records containing only components of allowed expressions. See also Section 3.7.3.2.
<code>sample(start, interval)</code>	Returns <code>true</code> and triggers time events at time instants <code>start + i*interval</code> ($i=0,1,\dots$). During continuous integration the operator returns always <code>false</code> . The starting time <code>start</code> and the sample interval <code>interval</code> need to be parameter expressions and need to be a subtype of <code>Real</code> or <code>Integer</code> .
<code>pre(y)</code>	Returns the “left limit” $y(t^{pre})$ of variable $y(t)$ at a time instant t . At an event instant, $y(t^{pre})$ is the value of y after the last event iteration at time instant t (see comment below). The <code>pre()</code> operator can be applied if the following three conditions are fulfilled simultaneously: (a) variable y is a subtype of a simple type, (b) y is a discrete-time expression (c) the operator is <i>not</i> applied in a function class. [Note: This can be applied to continuous-time variables in when-clauses, see Section 3.8.3 for the definition of discrete-time expression.] The first value of <code>pre(y)</code> is determined in the initialization phase. See also Section 3.7.3.1.
<code>edge(b)</code>	Is expanded into “ $(b \text{ and not } pre(b))$ ” for Boolean variable b . The same restrictions as for the <code>pre()</code> operator apply (e.g. not to be used in function classes).
<code>change(v)</code>	Is expanded into “ $(v <> pre(v))$ ”. The same restrictions as for the <code>pre()</code> operator apply.
<code>reinit(x, expr)</code>	In the body of a when clause, reinitializes x with <code>expr</code> at an event instant. x is a <code>Real</code> variable (resp. an array of <code>Real</code> variables, in which case vectorization applies according to Section 12.4.5) that must be selected as a state (resp., states) at least when the enclosing when clause becomes active.

`expr` needs to be type-compatible with `x`. The `reinit` operator can only be applied once for the same variable (resp. array of variables). It can only be applied in the body of a `when` clause. See also Section 8.3.6 .

A few of these operators are described in more detail in the following.

3.7.3.1 pre

A new event is triggered if at least for one variable v “`pre(v) <> v`” after the active model equations are evaluated at an event instant. In this case the model is at once reevaluated. This evaluation sequence is called “event iteration”. The integration is restarted, if for all v used in pre-operators the following condition holds: “`pre(v) == v`”.

[If v and `pre(v)` are only used in when-clauses, the translator might mask event iteration for variable v since v cannot change during event iteration. It is a “quality of implementation” to find the minimal loops for event iteration, i.e., not all parts of the model need to be reevaluated.]

The language allows mixed algebraic systems of equations where the unknown variables are of type Real, Integer, Boolean, or an enumeration. These systems of equations can be solved by a global fix point iteration scheme, similarly to the event iteration, by fixing the Boolean, Integer, and/or enumeration unknowns during one iteration. Again, it is a quality of implementation to solve these systems more efficiently, e.g., by applying the fix point iteration scheme to a subset of the model equations.]

3.7.3.2 noEvent and smooth

The `noEvent` operator implies that real elementary expressions are taken literally instead of generating crossing functions, Section 8.5. The `smooth` operator should be used instead of `noEvent`, in order to avoid events for efficiency reasons. A tool is free to not generate events for expressions inside `smooth`. However, `smooth` does not guarantee that no events will be generated, and thus it can be necessary to use `noEvent` inside `smooth`. *[Note that `smooth` does not guarantee a smooth output if any of the occurring variables change discontinuously.]*

[Example:

```
Real x,y,z;
parameter Real p;
equation
  x = if time<1 then 2 else time-2;
  z = smooth(0, if time<0 then 0 else time);
  y = smooth(1, noEvent(if x<0 then 0 else sqrt(x)*x));
  // noEvent is necessary.
```

]

3.8 Variability of Expressions

The concept of variability of an expression indicates to what extent the expression can vary over time. See also Section 4.4.4 regarding the concept of variability. There are four levels of variability of expressions, starting from the least variable:

- constant variability
- parameter variability
- discrete-time variability
- continuous-time variability

For an assignment $v := \text{expr}$ or binding equation $v = \text{expr}$, v must be declared to be at least as variable as `expr`.

- The binding equation of a parameter component and of the base type attributes [*such as* `start`] needs to be a parameter or constant expression.
- If v is a discrete-time component then `expr` needs to be a discrete-time expression.

3.8.1 Constant Expressions

Constant expressions are:

- Real, Integer, Boolean, String, and enumeration literals.
- Variables declared as constant.
- Except for the special built-in operators `initial`, `terminal`, `der`, `edge`, `change`, `sample`, and `pre` a function or operator with constant subexpressions as argument (and no parameters defined in the function) is a constant expression.

Components declared as constant shall have an associated declaration equation with a constant expression, if the constant is used in the model. The value of a constant cannot be changed after it has been given a value. A constant without an associated declaration equation can be given one by using a modifier.

3.8.2 Parameter Expressions

Parameter expressions are:

- Constant expressions.
- Variables declared as parameter.
- Except for the special built-in operators `initial`, `terminal`, `der`, `edge`, `change`, `sample` and `pre` a function or operator with parameter subexpressions is a parameter expression.

3.8.3 Discrete-Time Expressions

Discrete-time expressions are:

- Parameter expressions.
- Discrete-time variables, i.e., Integer, Boolean, String variables and enumeration variables, as well as Real variables assigned in when-clauses
- Function calls where all input arguments of the function are discrete-time expressions.
- Expressions where all the subexpressions are discrete-time expressions.
- Expressions in the body of a when-clause.
- Unless inside `noEvent`: Ordered relations (`>`, `<`, `>=`, `<=`) and the functions `ceil`, `floor`, `div`, `mod`, `rem`, `abs`, `sign`. These will generate events if at least one subexpression is not a discrete-time expression. [*This restriction guarantees that the `noEvent()` operator cannot be applied to Boolean, Integer, String, or enumeration equations outside of a when-clause, because then one of the two expressions is not discrete-time*]
- The functions `pre`, `edge`, and `change` result in discrete-time expressions.
- Expressions in functions behave as though they were discrete-time expressions.

For an equation `expr1 = expr2` where neither expression is of base type Real, both expressions must be discrete-time expressions. For record equations the equation is split into basic types before applying this test. [*This restriction guarantees that the `noEvent()` operator cannot be applied to Boolean, Integer, String, or enumeration equations outside of a when-clause, because then one of the two expressions is not discrete-time*]

Inside an if-expression, if-clause or for-clause, that is controlled by a non-discrete time switching expression and not in the body of a when-clause, it is not legal to have assignments to discrete variables, equations between discrete-time expressions, or real elementary relations/functions that should generate events. [*This restriction is necessary in order to guarantee that there all equations for discrete variable are discrete-time expressions, and to ensure that crossing functions do not become active between events.*]

[Example:

```
model Constants
  parameter Real p1 = 1;
  constant Real c1 = p1 + 2;    // error, no constant expression
  parameter Real p2 = p1 + 2;    // fine
end Constants;
```

```
model Test
  Constants c1(p1=3);    // fine
  Constants c2(p2=7);    // fine, declaration equation can be modified
  Boolean b;
  Real      x;
equation
  b = noEvent(x > 1) // error, since b is a discrete-time expr. and
                    // noEvent(x > 1) is not a discrete-time expr.
end Test;
]
```

3.8.4 Continuous-Time Expressions

All expressions are continuous-time expressions including constant, parameter and discrete expressions. The term “non-discrete-time expression” refers to expressions that are not constant, parameter or discrete expressions.

Chapter 4

Classes, Predefined Types, and Declarations

The fundamental structuring unit of modeling in Modelica is the class. Classes provide the structure for objects, also known as instances. Classes can contain equations which provide the basis for the executable code that is used for computation in Modelica. Conventional algorithmic code can also be part of classes. All data objects in Modelica are instantiated from classes, including the basic data types—`Real`, `Integer`, `String`, `Boolean`—and enumeration types, which are built-in classes or class schemata.

Declarations are the syntactic constructs needed to introduce classes and objects (i.e., components).

4.1 Access Control – Public and Protected Elements

Members of a Modelica class can have two levels of visibility: `public` or `protected`. The default is `public` if nothing else is specified

Protected elements in classes cannot be accessed via dot notation. They may not be modified or redeclared in a class modification.

All elements defined under the heading `protected` are regarded as protected. All other elements [*i.e.*, *defined under the heading `public`, without headings or in a separate file*] are public [*i.e.* *not protected*]. Regarding inheritance of protected and public elements, see Section 7.1.2.

4.2 Double Declaration not Allowed

The name of a declared element shall not have the same name as any other element in its partially flattened enclosing class. A component shall not have the same name as its type specifier. However, the internal flattening of a class can in some cases be interpreted as having two elements with the same name; these cases are described in Section 5.5, and Section 7.3.

4.3 Declaration Order and Usage before Declaration

Variables and classes can be used before they are declared.

[*In fact, declaration order is only significant for:*

- *Functions with more than one input variable called with positional arguments, Section 12.4.1.*
- *Functions with more than one output variable, Section 12.4.2.*
- *Records that are used as arguments to external functions, Section 12.8.1.3*
- *Enumeration literal order within enumeration types, Section 4.8.5.*

]

4.4 Component Declarations

Component declarations are described in this section.

4.4.1 Syntax and Examples of Component Declarations

The formal syntax of a component declaration clause is given by the following syntactic rules:

```
component_clause:
  type_prefix type_specifier [ array_subscripts ] component_list

type_prefix :
  [ flow ]
  [ discrete | parameter | constant ] [ input | output ]

type_specifier :
  name

component_list :
  component_declaration { , component_declaration }

component_declaration :
  declaration [ conditional_attribute ] comment

conditional_attribute:
  if expression

declaration :
  IDENT [ array_subscripts ] [ modification ]
```

[The declaration of a component states the type, access, variability, data flow, and other properties of the component. A component_clause i.e., the whole declaration, contains type prefixes followed by a type_specifier with optional array_subscripts followed by a component_list.]

There is no semantic difference between variables declared in a single declaration or in multiple declarations. For example, regard the following single declaration (component_clause) of two matrix variables:

```
Real[2,2] A, B;
```

That declaration has the same meaning as the following two declarations together:

```
Real[2,2] A;
Real[2,2] B;
```

The array dimension descriptors may instead be placed after the variable name, giving the two declarations below, with the same meaning as in the previous example:

```
Real A[2,2];
Real B[2,2];
```

The following declaration is different, meaning that the variable `a` is a scalar but `B` is a matrix as above:

```
Real a, B[2,2];
]
```

4.4.2 Component Declaration Static Semantics

If the type_specifier of the component declaration denotes a built-in type (RealType, IntegerType, etc.), the flattened or instantiated component has the same type.

If the type_specifier of the component does not denote a built-in type, the name of the type is looked up (Section 5.3). The found type is flattened with a new environment and the partially flattened enclosing class of the component. It is an error if the type is partial in a simulation model, or if a simulation model itself is partial. The new environment is the result of merging

- the modification of enclosing class element-modification with the same name as the component
- the modification of the component declaration

in that order.

Array dimensions shall be non-negative parameter expressions, or the colon operator denoting that the array dimension is left unspecified.

The rules for components in functions are described in Section 12.2.

Conditional declarations of components are described in Section 4.4.5.

4.4.2.1 Declaration Equations

An environment that defines the value of a component of built-in type is said to define a declaration equation associated with the declared component. For declarations of vectors and matrices, declaration equations are associated with each element. *[This makes it possible to override the declaration equation for a single element in an enclosing class modification, which would not be possible if the declaration equation is regarded as a single matrix equation.]*

4.4.2.2 Prefix Rules

Variables declared with the `flow` type prefix shall be a subtype of `Real`.

Type prefixes (i.e., `flow`, `discrete`, `parameter`, `constant`, `input`, `output`) shall only be applied for type, record and connector components – see also record specialized class, Section 4.6.

The type prefixes `flow`, `input` and `output` of a structured component are also applied to the elements of the component. The type prefixes `flow`, `input` and `output` shall only be applied for a structured component, if no element of the component has a corresponding type prefix of the same category. *[For example, `input` can only be used, if none of the elements has an `input` or `output` type prefix]*. The corresponding rules for the type prefixes `discrete`, `parameter` and `constant` are described in Section 4.4.4.1 for structured components.

The prefixes `input` and `output` have a slightly different semantic meaning depending on the context where they are used:

- In *functions*, these prefixes define the computational causality of the function body, i.e., given the variables declared as `input`, the variables declared as `output` are computed in the function body, see Section 12.4.
- In *simulation models* and *blocks* (i.e., on the top level of a model or block that shall be simulated), these prefixes define the interaction with the environment where the simulation model or block is used. Especially, the `input` prefix defines that values for such a variable have to be provided from the simulation environment and the `output` prefix defines that the values of the corresponding variable can be directly utilized in the simulation environment, see the notion of Globally balanced in Section 4.7.
- In component *models* and *blocks*, the `input` prefix defines that a binding equation has to be provided for the corresponding variable when the component is utilized in order to guarantee a locally balanced model (i.e., the number of local equations is identical to the local number of unknowns), see Section 4.7. Example:

```

block FirstOrder
    input Real u;
    ...
end FirstOrder;

model UseFirstOrder
    FirstOrder firstOrder(u=time); // binding equation for u
    ...
end UseFirstOrder;

```

The `output` prefix does not have a particular effect in a model or block component and is ignored.

- In *connectors*, prefixes `input` and `output` define that the corresponding connectors can only be connected according to block diagram semantics, see Section 9.1 (e.g., a connector with an `output` variable can only be connected to a connector where the corresponding variable is declared as `input`). There is the restriction that connectors which have at least one variable declared as `input` must be externally connected, see Section 4.7 (in order to get a locally balanced model, where the number of local unknowns is identical to the number of unknown equations). Together with the block diagram semantics rule this means, that such connectors must be connected *exactly once externally*.
- In *records*, prefixes `input` and `output` are not allowed, since otherwise a record could not be, e.g., passed as input argument to a function.

4.4.3 Acyclic Bindings of Constants and Parameters

The binding equations for parameters and constants in the translated model must be acyclic after flattening. Thus it is not possible to introduce equations for parameters by cyclic dependencies.

[Example:

```

constant Real p=2*q;
constant Real q=sin(p); // Illegal since p=2*q, q=sin(p) are cyclical

model ABCD
  parameter Real A[n,n];
  parameter Integer n=size(A,1);
end ABCD;

final ABCD a;
// Illegal since cyclic dependencies between size(a.A,1) and a.n

ABCD b(redeclare Real A[2,2]=[1,2;3,4]);
// Legal since size of A is no longer dependent on n.

ABCD c(n=2); // Legal since n is no longer dependent on the size of A.

```

]

4.4.4 Component Variability Prefixes *discrete*, *parameter*, *constant*

The prefixes *discrete*, *parameter*, *constant* of a component declaration are called variability prefixes and define in which situation the variable values of a component are initialized (see Section 8.5 and Section 8.6) and when they are changed in transient analysis (= solution of initial value problem of the hybrid DAE):

- A variable *vc* declared with the *parameter* or *constant* prefixes remains constant during transient analysis.
- A *discrete-time* variable *vd* has a vanishing time derivative (informally $\text{der}(\text{vd})=0$, but it is not legal to apply the $\text{der}()$ operator to discrete-time variables) and can change its values only at event instants during transient analysis (see Section 8.5).
- A *continuous-time* variable *vn* may have a non-vanishing time derivative ($\text{der}(\text{vn}) <> 0$ possible) and may also change its value discontinuously at any time during transient analysis (see Section 8.5). If there are any discontinuities the variable is not differentiable.

If a Real variable is declared with the prefix *discrete* it must in a simulation model be assigned in a when-clause, either by an assignment or an equation. The variable assigned in a when-clause may not be defined in a sub-component of model or block specialized class. [*This is to keep the property of balanced models*]

A Real variable assigned in a when-clause is a discrete-time variable, even though it was not declared with the prefix *discrete*. A Real variable not assigned in any when-clause and without any type prefix is a continuous-time variable.

The default variability for Integer, String, Boolean, or enumeration variables is discrete-time, and it is not possible to declare continuous-time Integer, String, Boolean, or enumeration variables. [*A Modelica translator is able to guarantee this property due to restrictions imposed on discrete expressions, see Section 3.8*]

The variability of expressions and restrictions on variability for definition equations is given in Section 3.8.

[*A discrete-time variable is a piecewise constant signal which changes its values only at event instants during simulation. Such types of variables are needed in order that special algorithms, such as the algorithm of Pantelides for index reduction, can be applied (it must be known that the time derivative of these variables is identical to zero). Furthermore, memory requirements can be reduced in the simulation environment, if it is known that a component can only change at event instants.*

A parameter variable is constant during simulation. This prefix gives the library designer the possibility to express that the physical equations in a library are only valid if some of the used components are constant during simulation. The same also holds for discrete-time and constant variables. Additionally, the parameter prefix allows a convenient graphical user interface in an experiment environment, to support quick changes of the most important constants of a compiled model. In combination with an if-clause, a parameter prefix allows to remove

parts of a model before the symbolic processing of a model takes place in order to avoid variable causalities in the model (similar to `#ifdef` in C). Class parameters can be sometimes used as an alternative. Example:

```

model Inertia
  parameter Boolean state = true;
  ...
equation
  J*a = t1 - t2;
  if state then          // code which is removed during symbolic
    der(v) = a;          // processing, if state=false
    der(r) = v;
  end if;
end Inertia;

```

A constant variable is similar to a parameter with the difference that constants cannot be changed after they have been given a value. It can be used to represent mathematical constants, e.g.

```

constant Real PI=4*arctan(1);

```

There are no continuous-time Boolean, Integer or String variables. In the rare cases they are needed they can be faked by using Real variables, e.g.:

```

  Boolean off1, off1a;
  Real off2;
equation
  off1 = s1 < 0;
  off1a = noEvent(s1 < 0); // error, since off1a is discrete
  off2 = if noEvent(s2 < 0) then 1 else 0; // possible
  u1 = if off1 then s1 else 0; // state events
  u2 = if noEvent(off2 > 0.5) then s2 else 0; // no state events

```

Since `off1` is a discrete-time variable, state events are generated such that `off1` is only changed at event instants. Variable `off2` may change its value during continuous integration. Therefore, `u1` is guaranteed to be continuous during continuous integration whereas no such guarantee exists for `u2`.

]

4.4.4.1 Variability of Structured Entities

For elements of structured entities with variability prefixes the most restrictive of the variability prefix and the variability of the component wins (using the default variability for the component if there is no variability prefix on the component).

[Example:

```

record A
  constant Real pi=3.14;
  Real y;
  Integer i;
end A;
parameter A a;
  // a.pi is a constant
  // a.y and a.i are parameters
  A b;
  // b.pi is a constant
  // b.y is a continuous-time variable
  // b.i is a discrete-time variable

```

]

4.4.5 Conditional Component Declaration

A component declaration can have a `condition_attribute`: "if" *expression*.

[Example:

```

  parameter Integer level=1;

```

```
Level1 component1(J=J) if level==1 "Conditional component";
Level component2 if level==2, component3 if level==3;
equation
  connect(component1..., ...) "Connection to conditional component";
  component1.u=0; // Illegal
]
```

The expression must be a Boolean scalar expression, and must be a parameter-expression [*that can be evaluated at compile time*].

If the Boolean expression is false the component is not present in the flattened DAE [*its modifier is ignored*], and connections to/from the component are removed. Other use of the component is illegal.

4.5 Class Declarations

Essentially everything in Modelica is a class, from the predefined classes `Integer` and `Real`, to large packages such as the Modelica standard library.

[*Example: A rather typical structure of a Modelica class is shown below. A class with a name, containing a number of declarations followed by a number of equations in an equation section.*]

```
class ClassName
  Declaration1
  Declaration2
  ...
equation
  equation1
  equation2
  ...
end ClassName;
]
```

The following is the formal syntax of class definitions, including the special variants described in later sections.

```
class_definition :
  [ encapsulated ]
  [ partial ]
  ( class | model | record | block | [ expandable ] connector | type |
    package | function )
  class_specifier
```

```
class_specifier :
  IDENT string_comment composition end IDENT
  | IDENT "=" base_prefix name [ array_subscripts ]
    [ class_modification ] comment
  | IDENT "=" enumeration "(" ( [enum_list] | ":" ) ")" comment
  | IDENT "=" der "(" name "," IDENT { "," IDENT } ")" comment
  | extends IDENT [ class_modification ] string_comment composition
    end IDENT
```

```
base_prefix :
  type_prefix
```

```
enum_list : enumeration_literal { "," enumeration_literal }
```

```
enumeration_literal : IDENT comment
```

```
composition :
  element_list
  { public element_list |
    protected element_list |
    equation_section |
    algorithm_section }
```

```

}
[ external [ language_specification ]
  [ external_function_call ] [ annotation ";" ]
  [ annotation ";" ] ]

```

4.5.1 Short Class Definitions

A class definition of the form

```
class IDENT1 = IDENT2 class_modification;
```

is identical, except for the lexical scope of modifiers, where the short class definition does not introduce an additional lexical scope for modifiers, to the longer form

```
class IDENT1
  extends IDENT2 class_modification;
end IDENT1;
```

[Example: demonstrating the difference in scopes:

```

model Resistor
  parameter Real R;
  ...
end Resistor;

model A
  parameter Real R;
  replaceable model Load=Resistor(R=R) constrainedby TwoPin;
  // Correct, sets the R in Resistor to R from model A.

  replaceable model LoadError
    extends Resistor(R=R);
    // Gives the singular equation R=R, since the right-hand side R
    // is searched for in LoadError and found in its base-class Resistor.
  end LoadError constrainedby TwoPin;
  Load a,b,c;
  ConstantSource ...;
  ...
end A;
]

```

A short class definition of the form

```
type TN = T[N] (optional modifier);
```

where N represents arbitrary array dimensions, conceptually yields an array class

```

'array' TN
  T[n] _ (optional modifiers);
'end' TN;

```

Such an array class has exactly one anonymous component (`_`). When a component of such an array class type is flattened, the resulting flattened component type is an array type with the same dimensions as `_` and with the optional modifier applied.

[Example:

```

type Force = Real[3] (unit={"Nm", "Nm", "Nm"});
Force f1;
Real f2[3] (unit={"Nm", "Nm", "Nm"});

```

the types of f1 and f2 are identical.]

If a short class definition inherits from a partial class the new class definition will be partial, regardless of whether it is declared with the keyword `partial` or not.

[Example:

```
replaceable model Load=TwoPin;
```

```

Load R; // Error unless Load is redeclared since TwoPin is a partial class.
]

```

If a short class definition does not specify any specialized class the new class definition will inherit the specialized class (this rule applies iteratively and also for redeclare).

A base-prefix applied in the short-class definition does not influence its type, but is applied to components declared of this type or types derived from it. It is not legal to combine other components with an extends from an array class, a class with non-empty base-prefix, or a simple type.

[Example:

```

type InArgument  = input Real;
type OutArgument = output Real[3];

function foo
  InArgument u; // Same as: input Real u
  OutArgument y; // Same as: output Real[3] y
algorithm
  y:=fill(u,3);
end foo;

Real x[:]=foo(time);
]

```

4.5.2 Local Class Definitions – Nested Classes

The local class should be statically flattenable with the partially flattened enclosing class of the local class apart from local class components that are partial or outer. The environment is the modification of any enclosing class element modification with the same name as the local class, or an empty environment.

The unflattened local class together with its environment becomes an element of the flattened enclosing class.

[The following example demonstrates parameterization of a local class:

```

class C1
  class Voltage = Real(nominal=1);
  Voltage v1, v2;
end C1;

class C2
  extends C1(Voltage(nominal=1000));
end C2;

```

Flattening of class C2 yields a local class Voltage with nominal-modifier 1000. The variables v1 and v2 are instances of this local class and thus have a nominal value of 1000.

]

4.6 Specialized Classes

Specialized kinds of classes [Earlier known as restricted classes] record, type, model, block, package, function, connector have the properties of a general class, apart from restrictions. Moreover, they have additional properties called enhancements. The following table summarizes the definition of the specialized classes:

record	Only public sections are allowed in the definition or in any of its components (i.e., equation, algorithm, initial equation, initial algorithm and protected sections are not allowed). May not be used in connections. The elements of a record may not have prefixes input, output, inner, outer, or flow. Enhanced with implicitly available record constructor function, see Section 12.6. Additionally, record components can be used as component references in expressions and in the left hand side of assignments, subject to normal type compatibility rules.
---------------	---

type	May only be predefined types, enumerations, array of type, or classes extending from type. Enhanced to extend from predefined types. [<i>No other specialized class has this property</i>]
model	Identical to <code>class</code> , the basic class concept, i.e., no restrictions and no enhancements.
block	Same as <code>model</code> with the restriction that each connector component of a <code>block</code> must have prefixes <code>input</code> and/or <code>output</code> for all connector variables. [<i>The purpose is to model input/output blocks of block diagrams. Due to the restrictions on input and output prefixes, connections between blocks are only possible according to block diagram semantic</i>]
package	May only contain declarations of classes and constants. Enhanced to allow <code>import</code> of elements of packages. (See also Chapter 13 on packages.)
function	See Section 12.2 for restrictions and enhancements of functions.
connector	No equations are allowed in the definition or in any of its components. Enhanced to allow <code>connect(..)</code> to components of connector classes.

4.7 Balanced Models

[In this section restrictions for `model` and `block` classes are present, in order that missing or too many equations can be detected and localized by a Modelica translator before using the respective `model` or `block` class. A non-trivial case is demonstrated in the following example:

```

partial model BaseCorrelation
  input Real x;
  Real y;
end BaseCorrelation;

model SpecialCorrelation // correct in Modelica 2.2 and 3.0
  extends BaseCorrelation(x=2);
equation
  y=2/x;
end SpecialCorrelation;

model UseCorrelation // correct according to Modelica 2.2
                     // not valid according to Modelica 3.0
  replaceable model Correlation=BaseCorrelation;
  Correlation correlation;
equation
  correlation.y=time;
end UseCorrelation;

model Broken // after redeclaration, there is 1 equation too much in Modelica 2.2
  UseCorrelation example(redeclare Correlation=SpecialCorrelation);
end Broken;

```

In this case one can argue that both `UseCorrelation` (adding an acausal equation) and `SpecialCorrelation` (adding a default to an input) are correct, but still when combined they lead to a model with too many equations – and it is not possible to determine which model is incorrect without strict rules, as the ones defined here.

In Modelica 2.2, `model Broken` will work with some models. However, by just redeclaring it to `model SpecialCorrelation`, an error will occur and it will be very difficult in a larger model to figure out the source of this error.

In Modelica 3.0, `model UseCorrelation` is no longer allowed and the translator will give an error. In fact, it is guaranteed that a redeclaration cannot lead to an unbalanced model any more.

].

The restrictions below apply after flattening – i.e. inherited components are included – possibly modified. The corresponding restrictions on connectors and connections are in Section 9.3.

Definition 1: Local Number of Unknowns

The local number of unknowns of a model or block class is the sum based on the components:

- For each declared component of specialized class `type` (Real, Integer, String, Boolean, enumeration and arrays of those, etc) or `record`, not declared as `outer`, it is the “number of unknown variables” inside it (i.e., excluding parameters and constants and counting the elements after expanding all records and arrays to a set of scalars of primitive types).
- Each declared component of specialized class `type` or `record` declared as `outer` is ignored [i.e., all variables inside the component are treated as known].
- For each declared component of specialized class `connector` component, it is the “number of unknown variables” inside it (i.e., excluding parameters and constants and counting the elements after expanding all records and arrays to a set of scalars of primitive types).
- For each declared component of specialized class `block` or `model`, it is the “sum of the number of inputs and flow variables” in the (top level) public connector components of these components (and counting the elements after expanding all records and arrays to a set of scalars of primitive types).

Definition 2: Local Equation Size

The local equation size of a model or block class is the sum of the following numbers:

- The number of equations defined locally (i.e. not in any model or block component), including binding equations, and equations generated from connect-equations. *This includes the proper count for when-clauses (see Section 8.3.5), and algorithms (see Section 11.1), and is also used for the flat Hybrid DAE formulation (see Appendix C).*
- The number of input and flow-variables present in each (top-level) public connector component. [This represents the number of connection equations that will be provided when the class is used.]
- The number of (top level) public input variables that neither are connectors nor have binding equations [i.e., top-level inputs are treated as known variables. This represents the number of binding equations that will be provided when the class is used.].

[To clarify top-level inputs without binding equation (for non-inherited inputs binding equation is identical to declaration equation, but binding equations also include the case where another model extends *M* and has a modifier on ‘*u*’ giving the value):

```

model M
  input Real u;
  input Real u2=2;
end M;

```

Here ‘*u*’ and ‘*u2*’ are top-level inputs and not connectors. The variable *u2* has a binding equation, but *u* does not have a binding equation. In the equation count, it is assumed that an equation for *u* is supplied when using the model.

]

Definition 3: Locally Balanced

A model or block class is “locally balanced” if the “local number of unknowns” is identical to the “local equation size” for all legal values of constants and parameters [respecting final bindings and min/max-restrictions. A tool shall verify the “locally balanced” property for the actual values of parameters and constants in the simulation model. It is a quality of implementation for a tool to verify this property in general, due to arrays of (locally) undefined sizes, conditional declarations, for loops etc].

Definition 4: Globally Balanced

Similarly as locally balanced, but including all unknowns and equations from all components. The global number of unknowns is computed by expanding all unknowns (i.e. excluding parameters and constants) into a set of scalars of primitive types. This should match the global equation size defined as:

- The number of equations defined (included in any model or block component), including equations generated from connect-equations.
- The number of input and flow-variables present in each (top-level) public connector component.
- The number of (top level) public input variables that neither are connectors nor have binding equations [*i.e., top-level inputs are treated as known variables*].

The following restrictions hold:

- In a non-partial model or block, all non-connector inputs of model or block components must have binding equations. [*E.g. if the model contains a component, firstOrder (of specialized class model) and firstOrder has 'input Real u' then there must be a binding equation for firstOrder.u.*]
- A component declared with the inner or outer prefix shall not be of a class having top-level public connectors containing inputs.
- Modifiers for components shall only contain redeclarations of replaceable elements and binding equations for parameters, constants (that do not yet have binding equations), inputs and variables having a default binding equation.
- **All non-partial model and block classes must be locally balanced** [*this means that the local number of unknowns equals the local equation size*].

Based on these restrictions, the following strong guarantee can be given for simulation models and blocks:

Proposition 1:

All simulation models and blocks are globally balanced.

[*Therefore the number of unknowns equal to the number of equations of a simulation model or block, provided that every used non-partial model or block class is locally balanced.*]

[*Example 1:*

```
connector Pin
  Real v;
  flow Real i;
end Pin;

model Capacitor
  parameter Real C;
  Pin p, n;
  Real u;
equation
  0 = p.i + n.i;
  u = p.v - n.v;
  C*der(u) = p.i;
end Capacitor;
```

Model Capacitor is a locally balanced model according to the following analysis:

Locally unknown variables: $p.i$, $p.v$, $n.i$, $n.v$, u

Local equations:

$$\begin{aligned} 0 &= p.i + n.i; \\ u &= p.v - n.v; \\ C \cdot \text{der}(u) &= p.i; \end{aligned}$$

and 2 equations corresponding to the 2 flow-variables $p.i$ and $n.i$.

*These are 5 equations in 5 unknowns (**locally balanced model**). A more detailed analysis would reveal that this is **structurally non-singular**, i.e. that the hybrid DAE will not contain a singularity independent of actual values.*

*If the equation " $u = p.v - n.v$ " would be missing in the Capacitor model, there would be 4 equations in 5 unknowns and the model would be **locally unbalanced** and thus simulation models in which this model is used would be usually structurally singular and thus not solvable.*

*If the equation " $u = p.v - n.v$ " would be replaced by the equation " $u = 0$ " and the equation $C \cdot \text{der}(u) = p.i$ would be replaced by the equation " $C \cdot \text{der}(u) = 0$ ", there would be 5 equations in 5 unknowns (**locally***

balanced), but the equations would be *singular*, regardless of how the equations corresponding to the flow-variables are constructed because the information that “*u*” is constant is given twice in a slightly different form.

Example 2:

```
connector Pin
  Real v;
  flow Real i;
end Pin;

partial model TwoPin
  Pin p,n;
end TwoPin;

model Capacitor
  parameter Real C;
  extends TwoPin;
  Real u;
equation
  0 = p.i + n.i;
  u = p.v - n.v;
  C*der(u) = p.i;
end Capacitor;

model Circuit
  extends TwoPin;
  replaceable TwoPin t;
  Capacitor c(C=12);
equation
  connect(p, t.p);
  connect(t.n, c.p);
  connect(c.n, n);
end Circuit;
```

Since *t* is partial we cannot check whether this is a **globally balanced model**, but we can check that *Circuit* is **locally balanced**.

Counting on model *Circuit* results in the following balance sheet:

Locally unknown variables (8): *p.i*, *p.v*, *n.i*, *n.v*, and 2 flow variables for *t* (*t.p.i*, *t.n.i*) and 2 flow variable for *c* (*c.p.i*, *c.n.i*).

Local equations:

$$\begin{aligned} p.v &= t.p.v; \\ 0 &= p.i - t.p.i; \\ c.p.v &= load.n.v; \\ 0 &= c.p.i + load.n.i; \\ n.v &= c.n.v; \\ 0 &= n.i - c.n.i; \end{aligned}$$

and 2 equation corresponding to the flow variables *p.i*, *n.i*

In total we have 8 scalar unknowns and 8 scalar equations, i.e., a **locally balanced model** (and this feature holds for any models used for the replaceable component “*t*”).

Some more analysis reveals that this local set of equations and unknowns is structurally non-singular. However, this does not provide any guarantees for the global set of equations, and specific combinations of models that are “locally non-singular” may lead to a globally non-singular model.]

Example 3:

```
import SI = Modelica.SIunits;

partial model BaseProperties
  "Interface of medium model for all type of media"
  parameter Boolean preferredMediumStates=false;
  constant Integer nXi "Number of independent mass fractions";
  InputAbsolutePressure      p;
  InputSpecificEnthalpy      h;
```



```

InputMassFraction      Xi[nXi];
SI.Temperature          T;
SI.Density              d;
SI.SpecificInternalEnergy u;

connector InputAbsolutePressure = input SI.AbsolutePressure;
connector InputSpecificEnthalpy = input SI.SpecificEnthalpy;
connector InputMassFraction = input SI.MassFraction;
end BaseProperties;

```

The use of connector here is a special design pattern. The variables p , h , X_i are marked as input to get correct equation count. Since they are connectors they should neither be given binding equations in derived classes nor when using the model. The design pattern is to give textual equations for them (as below); using connect-statements for these connectors would be possible (and would work) but is not part of the design.

This partial model defines that T, d, u can be computed from the medium model, provided p, h, X_i are given. Every medium with one or multiple substances and one or multiple phases, including incompressible media, has the property that T, d, u can be computed from p, h, X_i . A particular medium may have different “independent variables” from which all other intrinsic thermodynamic variables can be recursively computed. For example, a simple air model could be defined as:

```

model SimpleAir "Medium model of simple air. Independent variables: p,T"
  extends BaseProperties(nXi = 0,
    p(stateSelect = if preferredMediumStates then StateSelect.prefer
      else StateSelect.default),
    T(stateSelect = if preferredMediumStates then StateSelect.prefer
      else StateSelect.default));

  constant SI.SpecificHeatCapacity R = 287;
  constant SI.SpecificHeatCapacity cp = 1005.45;
  constant SI.Temperature T0 = 298.15
equation
  d = p / (R*T);
  h = cp*(T-T0);
  u = h - p/d;
end SimpleAir;

```

The local number of unknowns in model SimpleAir (after flattening) is:

- 3 (T, d, u : variables defined in BaseProperties and inherited in SimpleAir), plus
- $2+nX_i$ (p, h, X_i : variables inside connectors defined in BaseProperties and inherited in SimpleAir)

resulting in $5+nX_i$ unknowns. The local equation size is:

- 3 (equations defined in SimpleAir), plus
- $2+nX_i$ (input variables in the connectors inherited from BaseProperties)

Therefore, the model is locally balanced.

The generic medium model BaseProperties is used as a **replaceable** model in different components, like a dynamic volume or a fixed boundary condition:

```

import SI = Modelica.SIunits

connector FluidPort
  replaceable model Medium = BaseProperties;

  SI.AbsolutePressure p;
  flow SI.MassFlowRate m_flow;

  SI.SpecificEnthalpy h;
  flow SI.EnthalpyFlowRate H_flow;

  SI.MassFraction Xi [Medium.nXi] "Independent mixture mass fractions";
  flow SI.MassFlowRate mXi_flow[Medium.nXi] "Independent subst. mass flow rates";
end FluidPort;

model DynamicVolume

```

```

parameter SI.Volume V;
replaceable model Medium = BaseProperties;
FluidPort port(redeclare model Medium = Medium);
Medium medium(preferredMediumStates=true); // No modifier for p,h,Xi
SI.InternalEnergy U;
SI.Mass M;
SI.Mass MXi[medium.nXi];
equation
  U = medium.u*M;
  M = medium.d*V;
  MXi = medium.Xi*M;
  der(U) = port.H_flow; // Energy balance
  der(M) = port.m_flow; // Mass balance
  der(MXi) = port.mXi_flow; // Substance mass balance

// Equations binding to medium (inputs)
medium.p = port.p;
medium.h = port.h;
medium.Xi = port.Xi;
end DynamicVolume;

```

The local number of unknowns of DynamicVolume is:

- $4+2*n_{Xi}$ (inside the port connector), plus
- $2+n_{Xi}$ (variables U, M and MXi), plus
- $2+n_{Xi}$ (the input variables in the connectors of the medium model)

resulting in $8+4*n_{Xi}$ unknowns; the local equation size is

- $6+3*n_{Xi}$ from the equation section, plus
- $2+n_{Xi}$ flow variables in the port connector.

Therefore, DynamicVolume is a **locally balanced** model.

Note, when the DynamicVolume is used and the Medium model is redeclared to “SimpleAir”, then a tool will try to select p,T as states, since these variables have StateSelect.prefer in the SimpleAir model (this means that the default states U,M are derived quantities). If this state selection is performed, all intrinsic medium variables are computed from medium.p and medium.T, although p and h are the input arguments to the medium model. This demonstrates that in Modelica input/output does not define the computational causality. Instead, it defines that equations have to be provided here for p,h,Xi, in order that the equation count is correct. The actual computational causality can be different as it is demonstrated with the SimpleAir model.

```

model FixedBoundary_pTX
  parameter SI.AbsolutePressure p "Predefined boundary pressure";
  parameter SI.Temperature T "Predefined boundary temperature";
  parameter SI.MassFraction Xi[medium.nXi]
    "Predefined boundary mass fraction";

  replaceable model Medium = BaseProperties;
  FluidPort port(redeclare model Medium = Medium);
  Medium medium;
equation
  port.p = p;
  port.H_flow = semiLinear(port.m_flow, port.h, medium.h);
  port.MXi_flow = semiLinear(port.m_flow, port.Xi, medium.Xi);

// Equations binding to medium (note: T is not an input).
medium.p = p;
medium.T = T;
medium.Xi = Xi;
end FixedBoundary_pTX;

```

The number of local variables in FixedBoundary_pTX is:

- $4+2*n_{Xi}$ (inside the port connector), plus
- $2+n_{Xi}$ (the input variables in the connectors of the medium model)

resulting in $6+3*n_{Xi}$ unknowns, while the local equation size is

- $4+2*n_{Xi}$ from the equation section, plus
- $2+n_{Xi}$ flow variables in the port connector.

Therefore, FixedBoundary_pTX is a **locally balanced** model. The predefined boundary variables p and x_i are provided via equations to the input arguments `medium.p` and `medium.xi`, in addition there is an equation for T in the same way – even though T is not an input. Depending on the flow direction, either the specific enthalpy in the port (`port.h`) or h is used to compute the enthalpy flow rate H_{flow} . “ h ” is provided as binding equation to the medium. With the equation “`medium.T = T`”, the specific enthalpy “ h ” of the reservoir is indirectly computed via the medium equations. Again, this demonstrates, that an “input” just defines the number of equations have to be provided, but that it not necessarily defines the computational causality.

]

4.8 Predefined Types

The attributes of the predefined variable types and enumeration types are described below with Modelica syntax although they are predefined. Redclaration of any of these types is an error, and the names are reserved such that it is illegal to declare an element with these names. It is furthermore not possible to combine extends from the predefined types with other components. The definitions use `RealType`, `IntegerType`, `BooleanType`, `StringType`, `EnumType` as mnemonics corresponding to machine representations. [Hence the only way to declare a subtype of e.g. `Real` is to use the `extends` mechanism.]

4.8.1 Real Type

The following is the predefined `Real` type:

```
type Real // Note: Defined with Modelica syntax although predefined
  RealType value; // Accessed without dot-notation
  parameter StringType quantity = "";
  parameter StringType unit = "" "Unit used in equations";
  parameter StringType displayUnit = "" "Default display unit";
  parameter RealType min=-Inf, max=+Inf; // Inf denotes a large value
  parameter RealType start = 0; // Initial value
  parameter BooleanType fixed = true, // default for parameter/constant;
                                = false; // default for other variables
  parameter RealType nominal; // Nominal value
  parameter StateSelect stateSelect = StateSelect.default;
equation
  assert(value >= min and value <= max, "Variable value out of limit");
  assert(nominal >= min and nominal <= max, "Nominal value out of limit");
end Real;
```

4.8.2 Integer Type

The following is the predefined `Integer` type:

```
type Integer // Note: Defined with Modelica syntax although predefined
  IntegerType value; // Accessed without dot-notation
  parameter StringType quantity = "";
  parameter IntegerType min=-Inf, max=+Inf;
  parameter IntegerType start = 0; // Initial value
  parameter BooleanType fixed = true, // default for parameter/constant;
                                = false; // default for other variables
equation
  assert(value >= min and value <= max, "Variable value out of limit");
end Integer;
```

4.8.3 Boolean Type

The following is the predefined `Boolean` type:

```
type Boolean // Note: Defined with Modelica syntax although predefined
```

```
BooleanType value;                                // Accessed without dot-notation
parameter StringType quantity = "";
parameter BooleanType start = false; // Initial value
parameter BooleanType fixed = true, // default for parameter/constant;
                                     = false, // default for other variables
end Boolean;
```

4.8.4 String Type

The following is the predefined String type:

```
type String // Note: Defined with Modelica syntax although predefined
StringType value;                                // Accessed without dot-notation
parameter StringType quantity = "";
parameter StringType start = ""; // Initial value
end String;
```

4.8.5 Enumeration Types

A declaration of the form

```
type E = enumeration([enum_list]);
```

defines an enumeration type E and the associated enumeration literals of the enum-list. This is the only legal use of the enumeration keyword. The enumeration literals shall be distinct within the enumeration type. The names of the enumeration literals are defined inside the scope of E. Each enumeration literal in the `enum_list` has type E.

[Example:

```
type Size = enumeration(small, medium, large, xlarge);
Size t_shirt_size = Size.medium;
```

].

An optional comment string can be specified with each enumeration literal:

[Example:

```
type Size2 = enumeration(small "1st", medium "2nd", large "3rd", xlarge "4th");
```

]

An enumeration type is a simple type and the attributes are defined in Section 4.8.5.1. The Boolean type name or an enumeration type name can be used to specify the dimension range for a dimension in an array declaration and to specify the range in a for loop range expression. An element of an enumeration type can be accessed in an expression [e.g. *an array index value*].

[Example:

```
type DigitalCurrentChoices = enumeration(zero, one);
// Similar to Real, Integer
```

Setting attributes:

```
type DigitalCurrent = DigitalCurrentChoices(quantity="Current",
                                             start = one, fixed = true);
DigitalCurrent c(start = DigitalCurrent.one, fixed = true);
DigitalCurrentChoices c(start = DigitalCurrentChoices.one, fixed = true);
```

]

Accessing attribute values in expressions:

```
Real x[DigitalCurrentChoices];

// Example using the type name to represent the range

for e in DigitalCurrentChoices loop
  x[e] := 0.;
end loop;
```

```

for e loop          // Equivalent example using short form
  x[e] := 0.;
end loop;

// Equivalent example using the colon range constructor
for e in DigitalCurrentChoices.zero : DigitalCurrentChoices.one loop
  x[e] := 0.;
end loop;

model Mixing1 "Mixing of multi-substance flows, alternative 1"
  replaceable type E=enumeration(:)"Substances in Fluid";
  input  Real c1[E], c2[E], mdot1, mdot2;
  output Real c3[E], mdot3;
equation
  0 = mdot1 + mdot2 + mdot3;
  for e in E loop
    0 = mdot1*c1[e] + mdot2*c2[e] + mdot3*c3[e];
  end for;
  /* Array operations on enumerations are NOT (yet) possible:
     zeros(n) = mdot1*c1 + mdot2*c2 + mdot3*c3 // error
  */
end Mixing1;

model Mixing2 "Mixing of multi-substance flows, alternative 2"
  replaceable type E=enumeration(:)"Substances in Fluid";
  input  Real c1[E], c2[E], mdot1, mdot2;
  output Real c3[E], mdot3;
protected
  // No efficiency loss, since cc1, cc2, cc3
  // may be removed during translation
  Real cc1[:]=c1, cc2[:]=c2, cc3[:]=c3;
  final parameter Integer n = size(cc1,1);
equation
  0 = mdot1 + mdot2 + mdot3;
  zeros(n) = mdot1*cc1 + mdot2*cc2 + mdot3*cc3
end Mixing2;
]

```

4.8.5.1 Attributes of Enumeration Types

For each enumeration:

```
type E=enumeration(e1, e2, ..., en);
```

a new simple type is conceptually defined as

```

type E // Note: Defined with Modelica syntax although predefined
  EnumType value;          // Accessed without dot-notation
  parameter StringType quantity = "";
  parameter EnumType min=e1, max=en;
  parameter EnumType start = e1;    // Initial value
  parameter BooleanType fixed = true, // default for parameter/constant;
                                   = false; // default for other variables
  constant EnumType e1=...;
  ...
  constant EnumType en=...;
equation
  assert(value >= min and value <= max, "Variable value out of limit");
end E;

```

[Since the attributes and enumeration literals are on the same level, it is not possible to use the enumeration attribute names (quantity, min, max, start, fixed) as enumeration literals]

4.8.5.2 Type Conversion of Enumeration Values to String or Integer

The type conversion function `Integer(E.enumvalue)` returns the ordinal number of the enumeration value `E.enumvalue`, where `Integer(E.e1) = 1`, `Integer(E.en) = size(E)`, for an enumeration type `E=enumeration(e1, ..., en)`.

`String(E.enumvalue)` gives the string representation of the enumeration value. [Example: `String(E.Small)` gives "Small".]

See also Section 3.7.1.

4.8.6 Attributes start, fixed, and nominal

The attributes `start` and `fixed` define the initial conditions for a variable. “`fixed=false`” means an initial guess, i.e., value may be changed by static analyzer. “`fixed=true`” means a required value. The resulting consistent set of values for ALL model variables is used as initial values for the analysis to be performed.

The attribute `nominal` gives the nominal value for the variable. The user need not set it even though the standard does not define a default value. [The nominal value can be used by an analysis tool to determine appropriate tolerances or epsilons, or may be used for scaling. For example, the absolute tolerance for an integrator could be computed as “`absTol=abs(nominal)*relTol/100`”. A default value is not provided in order that in cases such as “`a=b`”, where “`b`” has a nominal value but not “`a`”, the nominal value can be propagated to the other variable).] [For external functions in C89, `RealType` by default maps to `double` and `IntegerType` by default maps to `int`. In the mapping proposed in Annex F of the C99 standard, `RealType/double` matches the IEC 60559:1989 (ANSI/IEEE 754-1985) double format. Typically `IntegerType` represents a 32-bit 2-complement signed integer.]

4.8.7 Other Predefined Types

4.8.7.1 StateSelect

The predefined `StateSelect` enumeration type is the type of the `stateSelect` attribute of the `Real` type. It is used to explicitly control state selection.

```
type StateSelect = enumeration(  
  never "Do not use as state at all.",  
  avoid "Use as state, if it cannot be avoided (but only if variable appears  
    differentiated and no other potential state with attribute  
    default, prefer, or always can be selected).",  
  default "Use as state if appropriate, but only if variable appears  
    differentiated.",  
  prefer "Prefer it as state over those having the default value  
    (also variables can be selected, which do not appear  
    differentiated). ",  
  always "Do use it as a state."  
);
```

4.8.7.2 ExternalObject

See Section 12.8.7 for information about the predefined type `ExternalObject`.

4.8.7.3 Graphical Annotation Types

A number of “predefined” record types and enumeration types for graphical annotations are described in Chapter 14. These types are not predefined in the usual sense since they cannot be referenced in ordinary Modelica code, only within annotations.

Chapter 5

Scoping, Name Lookup, and Flattening

This chapter describes the scope rules, and most of the name lookup and flattening of Modelica.

5.1 Flattening Context

Flattening is made in a context which consists of a modification environment (Section 7.2.2) and an ordered set of enclosing classes.

5.2 Enclosing Classes

The classes lexically enclosing an element form an ordered set of enclosing classes. A class defined inside another class definition (the enclosing class) precedes its enclosing class definition in this set.

Enclosing all class definitions is an unnamed enclosing class that contains all top-level class definitions, and not-yet read classes defined externally as described in Section 13.2.2. The order of top-level class definitions in the unnamed enclosing class is undefined.

During flattening, the enclosing class of an element being flattened is a partially flattened class. *[For example, this means that a declaration can refer to a name inherited through an extends-clause.]*

[Example:

```
class C1 ... end C1;
class C2 ... end C2;
class C3
  Real x=3;
  C1 y;
  class C4
    Real z;
  end C4;
end C3;
```

The unnamed enclosing class of class definition C3 contains C1, C2, and C3 in arbitrary order. When flattening class definition C3, the set of enclosing classes of the declaration of x is the partially flattened class C3 followed by the unnamed enclosing class with C1, C2, and C3. The set of enclosing classes of z is C4, C3 and the unnamed enclosing class in that order.]

5.3 Static Name Lookup

Names are looked up at class flattening to find names of base classes, component types, etc. Implicitly defined names of record constructor functions are ignored during type name lookup *[since a record and the implicitly created record constructor function, see Section 12.6, have the same name]*. Names of record classes are ignored during function name lookup.

5.3.1 Simple Name Lookup

For a simple name [not composed using dot-notation] lookup is performed as follows:

- First look for implicitly declared iteration variables if inside the body of a for-loop, Section 8.3.2 and Section 11.2.2, or if inside the body of a reduction expression, Section 10.3.4.
- When an element, equation, or section is flattened, any name is looked up sequentially in each member of the ordered set of enclosing classes until a match is found or an enclosing class is encapsulated. In the latter case the lookup stops except for the predefined types, functions and operators defined in this specification. For these cases the lookup continues in the global scope, where they are defined. *[E.g. abs is searched upwards in the hierarchy as usual. If an encapsulated boundary is reached, abs is searched in the global scope instead. The operator abs cannot be redefined in the global scope, because an existing class cannot be redefined at the same level.]*

Reference to variables successfully looked up in an enclosing class is only allowed for variables declared as constant. If the use is in a modifier on a short class definition, see Section 4.5.1.

[Example:

```
package A
  constant Real x=2;
  model B
    Real x;
    function foo
      output Real y
      algorithm y:=x; // Illegal since reference to non-constant x in B.
```

]

This lookup in each scope is performed as follows

- Among declared named elements (`class_definition` and `component_declaration`) of the class (including elements inherited from base-classes).
- Among the import names of qualified import statements in the lexical scope. The import name of `import A.B.C;` is C and the import name of `import D=A.B.C;` is D.
- Among the public members of packages imported via unqualified import-statements in the lexical scope. It is an error if this step produces matches from several unqualified imports.

[Note, that import statements defined in inherited classes are ignored for the lookup, i.e. import statements are not inherited.]

5.3.2 Composite Name Lookup

For a composite name of the form A.B [or A.B.C, etc.] lookup is performed as follows:

- The first identifier [A] is looked up as defined above.
- If the first identifier denotes a component, the rest of the name [e.g., B or B.C] is looked up among the declared named component elements of the component.
- If the identifier denotes a class, that class is temporarily flattened with an empty environment (i.e. no modifiers, see Section 7.2.2) and using the enclosing classes of the denoted class. The rest of the name [e.g., B or B.C] is looked up among the declared named elements of the temporary flattened class. If the class does not satisfy the requirements for a package, the lookup is restricted to encapsulated elements only. The class we look inside may not be partial in a simulation model.

[The temporary class flattening performed for composite names follow the same rules as class flattening of the base class in an extends-clause, local classes and the type in a component clause, except that the environment is empty. See also MoistAir2 example in Section 7.3 for further explanations regarding looking inside partial packages.]

5.3.3 Lookup of Imported Names

See Section 13.2.1.1.

5.4 Instance Hierarchy Name Lookup of Inner Declarations

An element declared with the prefix `outer` references an element instance with the same name but using the prefix `inner` which is nearest in the enclosing instance hierarchy of the outer element declaration.

There shall exist at least one corresponding inner element declaration for an outer element reference in a simulation model. *[Inner/outer components may be used to model simple fields, where some physical quantities, such as gravity vector, environment temperature or environment pressure, are accessible from all components in a specific model hierarchy. Inner components are accessible throughout the model, if they are not “shadowed” by a corresponding non-inner declaration in a nested level of the model hierarchy.]*

[Simple Example:

```
class A
  outer Real T0;
  ...
end A;

class B
  inner Real T0;
  A a1, a2;    // B.T0, B.a1.T0 and B.a2.T0 is the same variable
  ...
end B;
```

More complicated example:

```
class A
  outer Real TI;
  class B
    Real TI;
    class C
      Real TI;
      class D
        outer Real TI; //
      end D;
      D d;
    end C;
    C c;
  end B;
  B b;
end A;

class E
  inner Real TI;
  class F
    inner Real TI;
    class G
      Real TI;
      class H
        A a;
      end H;
      H h;
    end G;
    G g;
  end F;
  F f;
end E;

class I
  inner Real TI;
  E e;
```

```
// e.f.g.h.a.TI, e.f.g.h.a.b.c.d.TI, and e.f.TI is the same variable
// But e.f.TI, e.TI and TI are different variables
A a; // a.TI, a.b.c.d.TI, and TI is the same variable
end I;
]
```

The inner component shall be a subtype of the corresponding outer component. *[If the two types are not identical, the type of the inner component defines the instance and the outer component references just part of the inner component].*

[Example:

```
class A
  inner Real TI;
  class B
    outer Integer TI; // error, since A.TI is no subtype of A.B.TI
  end B;
end A;
]
```

5.4.1 Example of Field Functions using Inner/Outer

[Inner declarations can be used to define field functions, such as position dependent gravity fields, e.g.:

```
function A
  input Real u;
  output Real y;
end A;

function B // B is a subtype of A
  extends A;
algorithm
  ...
end B;

class C
  inner function fc = B; // define function to be actually used
  class D
    outer function fc = A;
    ...
  equation
    y = fc(u); // function B is used.
  end D;
end C;
]
```

5.5 Simultaneous Inner/Outer Declarations

An element declared with both the prefixes `inner` and `outer` conceptually introduces two declarations with the same name: one that follow the above rules for inner and another that follow the rules for outer. *[Local references for elements with both the prefix inner and outer references the outer element. That in turn references the corresponding element in an enclosing scope with the prefix inner.]*

Outer element declarations may only have modifications *[including declaration equations]* if they also have the inner prefix. Those modifications are only applied to the inner declaration.

[Example:

```
class A
  outer parameter Real p=2; // error, since modification
end A;
```

Intent of the following example: Propagate enabled through the hierarchy, and also be able to disable subsystems locally.

```

model ConditionalIntegrator "Simple differential equation if isEnabled"
  outer Boolean isEnabled;
  Real x(start=1);
equation
  der(x)=if isEnabled then -x else 0;
end ConditionalIntegrator;

model SubSystem "subsystem that 'enable' its conditional integrators"
  Boolean enableMe = time<=1;
  // Set inner isEnabled to outer isEnabled and enableMe
  inner outer Boolean isEnabled = isEnabled and enableMe;
  ConditionalIntegrator conditionalIntegrator;
  ConditionalIntegrator conditionalIntegrator2;
end SubSystem;

model System
  SubSystem subSystem;
  inner Boolean isEnabled = time>=0.5;
  // subSystem.conditionalIntegrator.isEnabled will be
  // 'isEnabled and subSystem.enableMe'
end System;
]

```

5.6 Flattening Process

In order to guarantee that elements can be used before they are declared and that elements do not depend on the order of their declaration (Section 4.3) in the enclosing class, the flattening proceeds in the following three steps, described in Section 5.6.1, Section 5.6.2, and Section 5.6.3, respectively.

5.6.1 Partial Flattening

First the names of declared local classes and components are found. Here modifiers are merged to the local elements and redeclarations take effect. Then base-classes are looked up, flattened and inserted into the class. The lookup of the base-classes should be independent of the order in which they are handled, and a name used for a base-class may not be inherited from any base-class.

[The lookup of the names of extended classes should give the same result before and after flattening the extends-clauses. One should not find any element used during this flattening by lookup through the extends-clauses. It should be possible to flatten all extends-clauses in a class before inserting the result of flattening. Local classes used for extends should be possible to flatten before inserting the result of flattening the extends-clauses.]

5.6.2 Flattening

Partially flatten the class, apply the modifiers (Section 7.2) and flatten all local elements.

5.6.3 Check of Flattening

Check that duplicate elements [*due to multiple inheritance*] are identical after flattening.

Chapter 6

Interface or Type Relationships

A class or component, e.g. denoted A, can in some cases be used at a location designed for another class or component, e.g. denoted B. In Modelica this is the case for replaceable classes (see Section 7.3) and for inner/outer elements (see Section 5.4). Replaceable classes are the primary mechanism to create very flexible models. In this chapter, the precise rules are defined when A can be used at a location designed for B. The restrictions are defined in terms of compatibility rules (Sections 6.3 and 6.4) between ”interfaces” (Section 6.1); this can also be viewed as sub-typing (Section 6.1).

In this chapter, two kinds of terminology is used for identical concepts to get better understanding [*e.g. by both engineers and computer scientists*]. A short summary of the terms is given in the following table. The details are defined in the rest of this chapter.

term	description
type or interface	The “essential” part of the public declaration sections of a class that is needed to decide whether A can be used instead of B [<i>E.g. a declaration “Real x” is part of the type, also called interface, but “import A” is not</i>].
class type or inheritance interface	The “essential” part of the public <i>and protected</i> declaration sections of a class that is needed to decide whether A can be used instead of B. The class type, also called inheritance interface, is needed when inheritance takes place, since then the protected declarations have to be taken into account.
subtype or compatible interface	A is a subtype of B, or equivalently, the interface of A is compatible to the interface of B, if the “essential” part of the public declaration sections of B is also available in A [<i>E.g., if B has a declaration “Real x”, this declaration must also be present in A. If A has a declaration “Real y”, this declaration must not be present in B</i>].
restricted subtype or plug compatible interface	A is a restricted subtype of B, or equivalently, the interface of A is plug compatible to the interface of B, if A is a subtype of B and if connector components in A that are not in B, are default connectable. [<i>E.g. it is not allowed that these connectors have variables with the “input” prefix, because then they must be connected.</i>] A model or block A cannot be used instead of B, if the particular situation does not allow to make a connection to these additional connectors. In such a case the stricter “plug compatible” is required for a redeclaration.
function subtype or function compatible interface	A is a function subtype of B, or equivalently, the interface of A is function compatible to the interface of B, if A is a subtype of B and if the additional arguments of function A that are not in function B are defined in such a way, that A can be called at places where B is called. [<i>E.g. an additional argument must have a default value.</i>]

6.1 The Concepts of Type, Interface and Subtype

A *type* can conceptually be viewed as a *set of values*. When we say that the variable x has the type `Real`, we mean that the value of x belongs to the set of values represented by the type `Real` i.e., roughly the set of floating point numbers representable by `Real`, for the moment ignoring the fact that `Real` is also viewed as a class with certain attributes. Analogously, the variable b having `Boolean` type means that the value of b belongs to the set of values $\{\text{false}, \text{true}\}$. The built-in types `Real`, `Integer`, `String`, `Boolean` are considered to be distinct types.

The *subtype* relation between types is analogous to the subset relation between sets. A type A_1 being a subtype of type A means that the set of values corresponding to type A_1 is a subset of the set of values corresponding to type A .

The type `Integer` is not a subtype of `Real` in Modelica even though the set of primitive integer values is a subset of the primitive real values since there are some attributes of `Real` that are not part of `Integer` (Section 4.7).

The concept of *interface* as defined in Section 6.2 and used in this document is equivalent to the notion of type based on sets in the following sense:

An element is characterized by its interface defined by some attributes (Section 6.2). The *type* of the element is the set of values having the same interface, i.e. the same attributes.

A *subtype* A_1 in relation to another type A , means that the elements of the set corresponding to A_1 is a subset of the set corresponding to A , characterized by the elements of that subset having additional properties.

[Example:

A record R : record R Boolean b ; Real x ; end R ;

Another record called R_2 : R_2 Boolean b ; Real x ; Real y ; end R_2 ;

An instance r : R r ;

An instance r_2 : R r_2 ;

The type R of r can be viewed as the set of all record values having the attributes defined by the interface of R , e.g. the infinite set $\{R(b=\text{false}, x=1.2), R(b=\text{false}, x=3.4), R(b=\text{true}, x=1.2), R(b=\text{true}, x=1.2, y=2), R(b=\text{true}, x=1.2, a=2), \dots\}$. The statement that r has the type (or interface) R means that the value of r belongs to this infinite set.

The type R_2 is a subtype of R since its instances fulfill the additional property of having the component `Real y`; in all its values.]

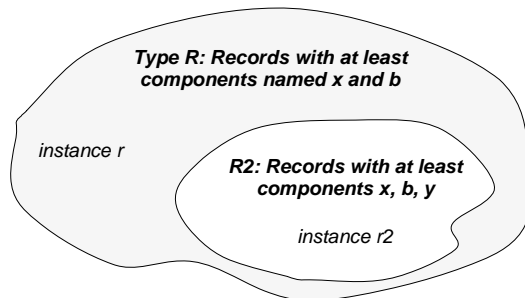


Figure 6-1. The type R can be defined as the set of record values containing x and b . The subtype R_2 is the subset of values that all contain x , b , and y .

]

6.2 Interface or Type

Based on a flattened class or component we can construct an interface for that flattened class or component. The interface or type [the terms *interface* and *type* are equivalent and can be used interchangeably] is defined as the following information about the flattened element itself:

- Whether it is replaceable or not.

- Whether the class itself or the class of the component is transitively non-replaceable (Section 6.2.1), and if not, the reference to replaceable class it refers to.
- Whether it is a component or a class.
- For each named public element of the class or component (including both local and inherited named elements) a tuple comprised of:
 - Name of the element.
 - Interface or type of the element. *This might have been modified by modifiers and is thus not necessarily identical to the interface of the original declaration.*
- Additional information/attributes about the element:
 - The `flow`-attribute.
 - Declared variability (`constant`, `parameter`, `discrete`).
 - The attributes `input` and `output`.
 - The attributes `inner` and/or `outer`.
 - Whether the declaration is `final`, and in that case its semantics contents.
 - Array sizes (if any).
 - Condition of conditional components (if any).
 - Which kind of specialized class.
 - For an enumeration type or component of enumeration type the names of the enumeration literals in order.
 - Whether it is a built-in type and the built-in type (`RealType`, `IntegerType`, `StringType` or `BooleanType`).

The corresponding *constraining* interface is constructed based on the *constraining* type (Section 7.3.2) of the declaration (if replaceable – otherwise same as actual type) and with the *constraining* interface for the named elements.

In a class all references to elements of that class should be limited to their constraining interface (*i.e. only public elements and if the declaration is replaceable limited to the constraining interface*).

[The public interface does not contain all of the information about the class or component. When using a class as a base-class we also need protected elements, and for internal type-checking we need e.g. import-elements. However, the information is sufficient for checking compatibility and for using the class to flatten components.]

6.2.1 Transitively non-Replaceable

[In several cases it is important that no new elements can be added to the interface of a class, especially considering short class definitions. Such classes are defined as transitively non-replaceable.]

A class reference is transitively non-replaceable iff (i.e. “if and only if”):

- If the class definition is long it is transitively non-replaceable if not declared replaceable.
- If the class definition is short (i.e. `class A=P.B`) it is transitively non-replaceable if it non-replaceable and equal to class reference (`“P.B”`) that is transitively non-replaceable.

[According to section 7.1.4, for a hierarchical name all parts of the name must be transitively non-replaceable, i.e. in “extends A.B.C” this implies that A.B.C must be transitively non-replaceable, as well as A and A.B, with the exception of the “class extends redeclaration mechanism” see section 7.3.1]

6.2.2 Inheritance Interface or Class Type

For inheritance the interface also must include protected elements; this is the only change compared to above.

Based on a flattened class we can construct an inheritance interface or class type for that flattened class. The inheritance interface or class type is defined as the following information about the flattened element itself:

- Whether it is replaceable or not.

- Whether the class itself or the class of the component is transitively non-replaceable (Section 6.2.1), and if not the reference to replaceable class it refers to.
- For each named element of the class (including both local and inherited named elements) a tuple comprised of:
 - Name of the element.
 - Whether the element is component or a class.
 - For elements that are classes: Inheritance interface or class type of the element. *This might have been modified by modifiers and is thus not necessarily identical to the interface of the original declaration.*
 - For elements that are components: interface or type of the element. *This might have been modified by modifiers and is thus not necessarily identical to the interface of the original declaration.*
- Additional information/attributes about the element:
 - The `flow`-attribute.
 - Declared variability (`constant`, `parameter`, `discrete`).
 - The attributes `input` and `output`.
 - The attributes `inner` and/or `outer`.
 - Whether the declaration is `final`, and in that case its semantics contents.
 - Array sizes (if any).
 - Condition of conditional components (if any).
 - Which kind of specialized class.
 - For an enumeration type or component of enumeration type the names of the enumeration literals in order.
 - Whether it is a built-in type and the built-in type (`RealType`, `IntegerType`, `StringType` or `BooleanType`).
 - Visibility (`public` or `protected`).

6.3 Interface Compatibility or Subtyping

An interface of a class or component A is compatible with an interface of a class or component B (or the constraining interface of B), or equivalently that the type of A is subtype of the type of B, iff [*intuitively all important elements of B must be present in*]:

- A is a class if and only if B is a class (and thus: A is a component if and only if B is a component).
- If B is not replaceable then A may not be replaceable.
- If B is transitively non-replaceable then A must be transitively non-replaceable (Section 6.2.1). For all elements of the inheritance interface of B there must exist a compatible element with the same name and visibility in the inheritance interface of A. The interface of A may not contain any other elements. [*We might even extend this to say that A and B should have the same contents, as in the additional restrictions below.*]
- If B is replaceable then for all elements of the component interface of B there must exist a plug-compatible element with the same name in the component interface of A.
- If B is neither transitively non-replaceable nor replaceable then A must be linked to the same class, and for all elements of the component interface of B there must thus exist a plug-compatible element with the same name in the component interface of A.
- Additional restrictions on the additional information. These elements should either match or have a natural total order:
 - If B is a non-replaceable long class definition A must also be a long class definition.
 - The flow-attribute should be matched for compatibility.
 - Variability is ordered `constant` < `parameter` < `discrete` < (no prefix: continuous-time for `Real`), and A is only compatible with B if the declared variability in A is less than or equal the variability in

B. *For a redeclaration of an element the variability attribute is as default inherited by the redeclaration (i.e. no need to repeat ‘parameter’ when redeclaring a parameter).*

- The input and output attributes must be matched. This ensures that the rules regarding inputs/outputs for matching connectors and (non-connector inputs) are preserved, as well as the restriction on blocks. *For a redeclaration of an element the input or output attribute is inherited from the original declaration.*
- The inner and/or outer attributes should be matched. *For a redeclaration of an element the inner and/or outer attributes are inherited from the original declaration (since it is not possible to have inner and/or outer attributes on a redeclare).*
- If B is final A must also be final and have the same semantic contents.
- The number of array dimensions in A and B must be matched. Furthermore the following must be valid for each array dimension: either the array size in B is unspecified (“:”) or the content of the array dimension in A is identical to the one in B.
- Conditional components are only compatible with conditional components. The conditions must have equivalent contents (similar as array sizes – except there is no “:” for conditional components). *For a redeclaration of an element the conditional part is inherited from the original.*
- A function class is only compatible with a function class, a package class only compatible with a package class, a connector class only with a connector class, a model or block class only compatible with a model or block class, and a type or record class only compatible with a type or record class.
- If B is an enumeration type A must also be an enumeration type and vice versa. If B is an enumeration type not defined as (:) then A must have the same enumeration literals in the same order.
- If B is a built-in type then A must also be of the same built-in type and vice versa.

Plug-compatibility is a further restriction of compatibility (subtyping) defined in Section 6.4, and further restricted for functions, see Section 6.5. For a replaceable declaration or modifier the default class must be compatible with the constraining class.

For a modifier the following must apply:

- The modified element should exist in the element being modified.
- The modifier should be compatible with the element being modified, and in most cases also plug-compatible, Section 6.4.

[If the original constraining flat class is legal (no references to unknown elements and no illegal use of class/component), and modifiers legal as above – then the resulting flat class will be legal (no references to unknown elements and no illegal use of class/component and compatible with original constraining class) and references refer to similar entities.]

6.4 Plug-Compatibility or Restricted Subtyping

[If a sub-component is redeclared, see Section 7.3, it is impossible to connect to any new connector. A connector with input attribute must be connected to, and since one cannot connect across hierarchies, one should not be allowed to introduce such a connector at a level where a connection is not possible. Therefore all public components present in the interface A that are not present in B must be connected by default.]

Definition 5: Plug-compatibility (= restricted subtyping)

An interface A is plug-compatible with (a restricted subtype of) an interface B (or the constraining interface of B) iff:

- A is compatible with (subtype of) B.
- All public components present in A but not in B must be default-connectable (as defined below).

Definition 6: Default connectable

A component of an interface is default-connectable iff:

- All of its components are default connectable.
- A connector component must not be an input. *[Otherwise a connection to the input will be missing.]*
- A connector component must not be of an expandable connector class. *[The expandable connector does potentially have inputs.]*
- A parameter, constant, or non-connector input must either have a binding equation or all of its sub-components must have binding equations.

Based on the above definitions, there are the following restrictions:

- A redeclaration of an inherited top-level component must be *compatible with* (subtype of) the constraining interface of the element being redeclared.
- In all other cases redeclarations must be *plug-compatible* with the constraining interface of the element being redeclared.

[The reason for the difference is that for an inherited top-level component it is possible to connect to the additional connectors, either in this class or in a derived class.]

Example:

```
partial model TwoFlanges
  Modelica.Mechanics.Rotational.Interfaces.Flange_a flange_a;
  Modelica.Mechanics.Rotational.Interfaces.Flange_b flange_b;
end TwoFlanges;

partial model FrictionElement
  extends TwoFlanges;
...
end FrictionElement;

model Clutch "compatible - but not plug-compatible with FrictionElement"
  Modelica.Blocks.Interfaces.RealInput pressure;
  extends FrictionElement;
...
end Clutch;

model DriveLineBase
  extends TwoFlanges;
  Inertia J1;
  replaceable FrictionElement friction;
equation
  connect(flange_a, J1.flange_a);
  connect(J1.flange_b, friction.flange_a);
  connect(friction.flange_b, flange_b);
end DriveLineBase;

model DriveLine
  extends DriveLineBase(redeclare Clutch friction);
  Constant const;
equation
  connect(const.y, friction.pressure);
  // Legal connection to new input connector.
end DriveLine;

model UseDriveLine "illegal model"
  DriveLineBase base(redeclare Clutch friction);
  // Cannot connect to friction.pressure
end UseDriveLine;
```

If a subcomponent is redeclared, it is impossible to connect to any new connector. Thus any new connectors must work without being connected, i.e., the default connection of flow-variables. That fails for inputs (and expandable connectors may contain inputs). For parameters and non-connector inputs it would be possible to provide

bindings in a derived class, but that would require hierarchical modifiers and it would be bad modeling practice that a hierarchical modifier must be used in order to make a model valid. A replaceable class might be used as the class for a sub-component, therefore plug-compatibility is required not only for replaceable sub-components, but also for replaceable classes.]

6.5 Function-Compatibility or Function-Subtyping for Functions

[Functions may be called with either named or positional arguments, and thus both the name and order is significant. If a function is redeclared, see Section 7.3, any new arguments must have defaults (and be at the end) in order to preserve the meaning of existing calls.]

Definition 7: Function-Compatibility or Function-Subtyping for Functions

A function class A is *function-compatible* with or a *function subtype* of function class B iff, *[The terms function-compatible and function subtype of are synonyms and used interchangeably]*:

- A is compatible to (subtype of) B.
- All public input components of B have correspondingly named public input components of A in the same order and preceding any additional public input components of A.
- All public output components of B have correspondingly named public output components of A in the same order and preceding any additional public output components of A.
- A public input component of A must have a binding assignment if the corresponding named element has a binding assignment in B.
- A public input component of A not present in B must have a binding assignment.

Based on the above definition the following restriction holds:

- The interface of a redeclared function must be *function-compatible with or a function subtype of* the constraining interface of the function being redeclared.

[Example: Demonstrating a redeclaration using a function-compatible function]

```
function GravityInterface
  input Modelica.SIunits.Position position[3];
  output Modelica.SIunits.Acceleration acceleration[3];
end GravityInterface;

function PointMassGravity
  extends GravityInterface;
  input Modelica.SIunits.Mass m;
algorithm
  acceleration:= -Modelica.Constants.g*m*position/(position*position)^1.5;
end PointMassGravity;

model Body
  Modelica.Mechanics.MultiBody.Interface.Frame_a frame_a;
  replaceable function gravity=GravityInterface;
equation
  frame_a.f = gravity(frame_a.r0); // or gravity(position=frame_a.r0);
  frame_a.t = zeros(3);
end Body;

model PlanetSimulation
  function sunGravity = PointMassGravity(m=2e30);
  Body planet1(redeclare function gravity=sunGravity);
  Body planet2(redeclare function gravity=PointMassGravity(m=2e30));
  ...
end PlanetSimulation;
```

Note: PointMassGravity is not function-compatible with GravityInterface (no default for m), but sunGravity inside PlanetSimulation is function-compatible with GravityInterface.]

6.6 Type Compatible Expressions

Certain expressions consist of an operator applied to two or more type compatible sub-expressions (A and B), including binary operators, e.g. $A + B$, if-expressions, e.g. `if x then A else B`, and array expressions, e.g. $\{A, B\}$. The resulting type of the expression in case of two type compatible subexpressions A and B is defined as follows:

- If A is a record-expression B must also be a record-expression with the same named elements. The type compatible expression is a record comprised of named elements that are compatible with the corresponding named elements of both A and B.
- If A is an array expression then B must also be an array expression, and $\text{ndims}(A) = \text{ndims}(B)$. The type compatible expression is an array expression with elements compatible with the elements of both A and B. If both $\text{size}(A)$ and $\text{size}(B)$ are known and $\text{size}(A) = \text{size}(B)$ then this defines the size of the type compatible expression, otherwise the size of the expression is not known until the expression is about to be evaluated. In case of an if-expression the size of the type compatible expression is defined based on the branch selected, and for other cases $\text{size}(A) = \text{size}(B)$ must hold at this point..
- If A is a scalar expression of a simple type B must also be a scalar expression of a simple type.
- If A is a Real expression then B must be a Real or Integer expression and the type compatible expression is Real.
- If A is an Integer expression then B must be a Real or Integer expression and the type compatible expression is Real or Integer (same as B).
- If A is a Boolean expression then B must be a Boolean expression and the type compatible expression is Boolean.
- If A is a String expression then B must be a String expression and the type compatible expression is String.
- If A is an enumeration expression then B must be an enumeration expression and the type compatible expression is enumeration expression, and all enumeration expressions must be defined in terms of an enumeration type with the same enumeration literals in the same order.

Chapter 7

Inheritance, Modification, and Redeclaration

One of the major benefits of object-orientation is the ability to *extend* the behavior and properties of an existing class. The original class, known as the *base class*, is extended to create a more specialized version of that class, known as the *derived class* or *subclass*. In this process, the data and behavior of the original class in the form of variable declarations, equations, and certain other contents are reused, or *inherited*, by the subclass. In fact, the inherited contents is copied from the superclass into the subclass, but before copying certain operations, such as type expansion, checking, and modification, are performed on the inherited contents when appropriate. This chapter describes the inheritance concept in Modelica, together with the related concepts modification and redeclaration.

7.1 Inheritance—Extends Clause

The extends-clause is used to specify inheritance from a base class into an (enclosing) class containing the extends-clause. The syntax of the extends-clause is as follows:

```
extends_clause :
  extends name [ class_modification ] [annotation]
```

The name of the base class is looked up in the partially flattened enclosing class (Section 5.2) of the extends-clause. The found base class is flattened with a new environment and the partially flattened enclosing class of the extends-clause. The new environment is the result of merging

- arguments of all enclosing class environments that match names in the flattened base class
- the optional `class_modification` of the extends-clause

in that order.

[Examples of the three rules are given in the following example:

```
class A
  parameter Real a, b;
end A;

class B
  extends A(b=2);      // Rule #2
end B;

class C
  extends B(a=1);      // Rule #1
end C;
]
```

The elements of the flattened base class become elements of the flattened enclosing class [*e.g., including equation sections and algorithm sections, but excluding import-clauses*].

[From the example above we get the following flattened class:

```

class Cinstance
  parameter Real a=1;
  parameter Real b=2;
end Cinstance;

```

The ordering of the merging rules ensures that, given classes *A* and *B* defined above,

```

class C2
  B bcomp(b=3);
end C2;

```

yields an instance with `bcomp.b=3`, which overrides `b=2`.]

The declaration elements of the flattened base class shall either

- Not already exist in the partially flattened enclosing class [*i.e.*, have different names] .
- The new element is a long form of redeclare or uses the ‘class extends A’ syntax, see Section 7.3.
- Be exactly identical to any element of the flattened enclosing class with the same name and the same level of protection (public or protected) and same contents. In this case, one of the elements is ignored (since they are identical it does not matter which one).

Otherwise the model is incorrect.

Equations of the flattened base class that are syntactically equivalent to equations in the flattened enclosing class are discarded. [Note: equations that are mathematically equivalent but not syntactically equivalent are not discarded, hence yield an overdetermined system of equations.]

7.1.1 Multiple Inheritance

Multiple inheritance is possible since multiple extends-clauses can be present in a class.

7.1.2 Inheritance of Protected and Public Elements

If an extends-clause is used under the `protected` heading, all elements of the base class become protected elements of the current class. If an extends-clause is a public element, all elements of the base class are inherited with their own protection. The eventual headings `protected` and `public` from the base class do not affect the consequent elements of the current class (*i.e.*, headings `protected` and `public` are not inherited).

7.1.3 Restrictions on the Kind of Base Class

Since specialized classes of different kinds have different properties, see Section 4.6, only specialized classes that are “in some sense compatible” to each other can be derived from each other via inheritance. The following table shows which kind of specialized class can be used in an extends clauses of another kind of specialized class (the “grey” cells mark the few exceptional cases, where a specialized class can be derived from a specialized class of another kind):

Derived Class	Base Class						
	package	function	type	record	connector	block	model
package	yes						
function		yes					
type			yes				
record				yes			
connector			yes	yes	yes		
block				yes		yes	
model				yes		yes	yes

The specialized classes `package`, `function`, `type` and `record` can only be derived from their own kind [(e.g. a package can only be base class for packages. All other kinds of classes can use the `import` statement to use the contents of a package)].

[Examples:

```

record RecordA
  ...
end RecordA;

package PackageA
  ...
end PackageA;

package PackageB
  extends PackageA; // fine
end PackageB;

model ModelA
  extends RecordA; // fine
end ModelA;

model ModelB
  extends PackageA; // error, inheritance not allowed
end ModelB;
]
```

7.1.4 Restrictions on Base Classes and Constraining Types to be Transitively Non-Replaceable

The class name used after `extends` for base-classes and for constraining classes must use a class reference considered transitively non-replaceable, i.e. there are no replaceable elements in the referenced class, or any of its base classes or constraining types transitively at any level, see definition in Section 6.2.1.. [This formulation excludes the long form of `redeclare`, i.e. ‘`redeclare model extends M...`’ where `M` must be an inherited replaceable class.] For a replaceable component declaration without constraining clause the class must use a class reference considered transitively non-replaceable. [This implies that constraining classes are always transitively non-replaceable – both if explicitly given or implicitly by the declaration.]

7.2 Modifications

There are three kinds of constructs in the Modelica language in which modifications can occur:

- Variable declarations.
- Short class declarations.
- Extends-clauses.

A modifier modifies one or more declarations from an inherited class by changing some aspect(s) of the inherited declarations. The most common kind of modifier just changes the *default value* or the *start value* in a binding equation.

[Example: Modifying the default start value of the altitude variable:

```

Real altitude(start= 59404);
]
```

A more dramatic change is to modify the *type* and/or the *prefixes* and possibly the *dimension sizes* of a declared element. This kind of modification is called a *redeclaration* (Section 7.3) and requires the special keyword

`redeclare` to be used in the modifier in order to reduce the risk for accidental modeling errors. In most cases a declaration that can be redeclared must include the prefix `replaceable` (Section 7.3)

7.2.1 Syntax of Modifications and Redeclarations

The syntax is as follows:

```
modification :
  class_modification [ "=" expression ]
| "=" expression
| "!=" expression

class_modification :
  "(" [ argument_list ] ")"

argument_list :
  argument { ",", argument }

argument :
  element_modification_or_replaceable
| element_redeclaration

element_modification_or_replaceable:
  [ each ] [ final ] ( element_modification | element_replaceable )

element_modification :
  component_reference [ modification ] string_comment

element_redeclaration :
  redeclare [ each ] [ final ]
( ( class_definition | component_clause1 ) | element_replaceable )

element_replaceable:
  replaceable ( class_definition | component_clause1 )
  [constraining_clause]

component_clause1 :
  type_prefix type_specifier component_declaration1

component_declaration1 :
  declaration comment
```

7.2.2 Modification Environment

The modification environment contains arguments which modify elements of the class (e.g., parameter changes). The modification environment is built by merging class modifications, where outer modifications override inner modifications. *[Note: this should not be confused with inner outer prefixes described in Section 5.4]*

7.2.3 Merging of Modifications

Merging of modifiers means that outer modifiers override inner modifiers. The merging is hierarchical, and a value for an entire non-simple overrides value modifiers for all components, and it is an error if this overrides a final attribute for a component. When merging modifiers each modification keeps its own each-attribute.

[The following larger example demonstrates several aspects:

```
class C1
  class C11
    parameter Real x;
  end C11;
end C1;
class C2
```



```

class C21
...
end C21;
end C2;
class C3
  extends C1;
  C11 t(x=3);           // ok, C11 has been inherited from C1
  C21 u;                 // ok, even though C21 is inherited below
  extends C2;
end C3;

```

The modification environment of the declaration of `t` is `(x=3)`. The modification environment is built by merging class modifications, as shown by:

```

class C1
  parameter Real a;
end C1;
class C2
  parameter Real b,c;
end C2;
class C3
  parameter Real x1;           // No default value
  parameter Real x2 = 2;      // Default value 2
  parameter C1 x3;            // No default value for x3.a
  parameter C2 x4(b=4);       // x4.b has default value 4
  parameter C1 x5(a=5);       // x5.a has default value 5
  extends C1;                 // No default value for inherited element a
  extends C2(b=6,c=77);       // Inherited b has default value 6
end C3;
class C4
  extends C3(x2=22, x3(a=33), x4(c=44), x5=x3, a=55, b=66);
end C4;

```

Outer modifications override inner modifications, e.g., `b=66` overrides the nested class modification of `extends C2(b=6)`. This is known as merging of modifications: `merge((b=66), (b=6))` becomes `(b=66)`.

A flattening of class `C4` will give an object with the following variables:

Variable	Default value
<code>x1</code>	<code>none</code>
<code>x2</code>	<code>22</code>
<code>x3.a</code>	<code>33</code>
<code>x4.b</code>	<code>4</code>
<code>x4.c</code>	<code>44</code>
<code>x5.a</code>	<code>x3.a</code>
<code>a</code>	<code>55</code>
<code>b</code>	<code>66</code>
<code>c</code>	<code>77</code>

]

7.2.4 Single Modification

Two arguments of a modification shall not designate the same primitive attribute of an element. When using qualified names the different qualified names starting with the same identifier are merged into one modifier.

[Example:

```

class C1
  Real x[3];
end C1;
class C2 = C1(x=ones(3), x[2]=2); // Error: x[2] designated twice
class C3
  class C4
    Real x;

```

```
end C4;  
C4 a(x.unit = "V", x.displayUnit="mV", x=5.0);  
// Ok, different attributes designated (unit, displayUnit and value)  
// identical to:  
C4 b(x(unit = "V", displayUnit="mV") = 5.0));  
end C3;  
]
```

7.2.5 Modifiers for Array Elements

The following holds:

- The `each` keyword on a modifier requires that it is applied in an array declaration/modification, and the modifier is applied individually to each element of the array. If the modified element is a vector and the modifier does not contain the `each`-attribute, the modification is split such that the first element in the vector is applied to the first element of the vector of elements, the second to the second element, etc. Matrices and general arrays of elements are treated by viewing those as a vectors of vectors etc.
- If the modified element is a vector with subscripts the subscripts must be Integer literals.
- If a nested modifier is split, the split is propagated to all elements of the nested modifier, and if they are modified by the `each`-keyword the split is inhibited for those elements. If the nested modifier that is split in this way contains re-declarations that are split it is illegal.

[Example:

```
model C  
  parameter Real a [3];  
  parameter Real d;  
end C;  
  
model B  
  C c[5] (each a = {1,2,3}, d={1,2,3,4,5});  
end B;
```

This implies that $c[i].a[j]=j$, and $c[i].d=i$.

]

7.2.6 Final Element Modification Prevention

An element defined as final by the `final` prefix in an element modification or declaration cannot be modified by a modification or by a redeclaration. All elements of a final element are also final. [Setting the value of a parameter in an experiment environment is conceptually treated as a modification. This implies that a final modification equation of a parameter cannot be changed in a simulation environment].

[Examples:

```
type Angle = Real(final quantity="Angle", final unit = "rad",  
                  displayUnit="deg");  
Angle a1(unit="deg");           // error, since unit declared as final!  
Angle a2(displayUnit="rad");    // fine  
  
model TransferFunction  
  parameter Real b[:] = {1} "numerator coefficient vector";  
  parameter Real a[:] = {1,1} "denominator coefficient vector";  
  ...  
end TransferFunction;  
  
model PI "PI controller";  
  parameter Real k=1 "gain";  
  parameter Real T=1 "time constant";  
  TransferFunction tf(final b=k*{T,1}, final a={T,0});  
end PI;
```

```

model Test
  PI c1(k=2, T=3);    // fine
  PI c2(b={1});       // error, b is declared as final
end Test;
]

```

7.3 Redeclaration

A `redeclare` construct in a modifier replaces the declaration of a local class or component in the modified element with another declaration. A `redeclare` construct as an element replaces the inherited declaration of a local class or component with another declaration.

A modifier with the keyword `replaceable` is automatically seen as being a `redeclare`.

In redeclarations some parts of the original declaration is automatically inherited by the new declaration. This is intended to make it easier to write declarations by not having to repeat common parts of the declarations, and does in particular apply to attributes that must be identical. The inheritance only applies to the declaration itself and not to elements of the declaration.

The general rule is that if no attribute within one of the following groups is present in the new declaration the old attributes of that kind are preserved.

The groups that are valid for both classes and components:

- `public`, `protected`
- `inner`, `outer`
- constraining type according to rules in Section 7.3.2.

The groups that are only valid for components:

- `flow`
- `discrete`, `parameter`, `constant`
- `input`, `output`
- array dimensions

[Note: The inheritance is from the original declaration. In most cases replaced or original does not matter. It does matter if a user redeclares a variable to be a parameter and then redeclares it without parameter.]

[Example:

```

model HeatExchanger
  replaceable parameter GeometryRecord geometry;
  replaceable input Real u[2];
end HeatExchanger;

HeatExchanger(
  /*redeclare*/ replaceable /*parameter */ GeoHorizontal geometry,
  redeclare /* input */ Modelica.SIunits.Angle u /*[2]*/);
// The semantics ensure that parts in /*.*/ are automatically added
// from
]

```

7.3.1 The class extends Redeclaration Mechanism

A class declaration of the type '`class extends B(...)`', where `class` as usual can be replaced by any other specialized class, replaces the inherited class `B` with another declaration that extends the inherited class where the optional class-modification is applied to the inherited class. *[Since this implies that all declarations are inherited with modifications applied there is no need to apply modifiers to the new declaration.]*

For '`class extends B(...)`' the inherited class is subject to the same restrictions as a `redeclare` of the inherited element and the new element is only `replaceable` if the new definition is `replaceable`. It is not subject to the restriction that `B` should be transitively non-`replaceable` (in fact `B` should be `replaceable`).

The syntax rule for `class` extends construct is in the definition of the `class_specifier` nonterminal (see also class declarations in Section 4.5):

```
class_definition :
  [ encapsulated ]
  [ partial ]
  ( class | model | record | block | [ expandable ] connector | type |
    package | function )
  class_specifier

class_specifier :
  ...
  | extends IDENT [ class_modification ] string_comment composition
```

The nonterminal `class_definition` is referenced in several places in the grammar, including the following case which is used in some examples below, including `package extends` and `model extends`:

```
element :
  import_clause |
  extends_clause |
  [ redeclare ]
  [ final ]
  [ inner ] [ outer ]
  ( ( class_definition | component_clause ) |
    replaceable ( class_definition | component_clause )
    [constraining_clause comment])
```

[Example:

```
class A
  parameter Real x;
end A;

class B
  parameter Real x=3.14, y;    // B is a subtype of A
end B;

class C
  replaceable A a(x=1);
end C;

class D
  extends C(redeclare B a(y=2));
end D;
```

which effectively yields a class D2 with the contents

```
class D2
  B a(x=1, y=2);
end D2;
```

Example to extend from existing packages:

```
package PowerTrain    // library from someone else
  replaceable package GearBoxes
  ...
end GearBoxes;
end PowerTrain;

package MyPowerTrain
  extends PowerTrain;  // use all classes from PowerTrain

  package extends Gearboxes  // add classes to sublibrary
  ...
end Gearboxes;
end MyPowerTrain;
```

Example for an advanced type of package structuring with constraining types:

```
partial package PartialMedium "Generic medium interface"
  constant Integer nX "number of substances";
```

```

replaceable partial model BaseProperties
  Real X[nX];
  ...
end BaseProperties;

replaceable partial function dynamicViscosity
  input Real p;
  output Real eta;...
end dynamicViscosity;
end PartialMedium;

package MoistAir "Special type of medium"
  extends PartialMedium(nX = 2);

  redeclare model extends BaseProperties (T(stateSelect=StateSelect.prefer))
    // replaces BaseProperties by a new implementation and
    // extends from Baseproperties with modification
    // note, nX = 2 (!)
  equation
    X = {0, 1};
    ...
  end BaseProperties

  redeclare function extends dynamicViscosity
    // replaces dynamicViscosity by a new implementation and
    // extends from dynamicViscosity
  algorithm
    eta = 2*p;
  end dynamicsViscosity;
end MoistAir;

```

Note, since MostAir extends from PartialMedium, constant nX=2 in package MoistAir and the model BaseProperties and the function dynamicViscosity is present in MoistAir. By the following definitions, the available BaseProperties model is replaced by another implementation which extends from the BaseProperties model that has been temporarily constructed during the extends of package MoistAir from PartialMedium. The redeclared BaseProperties model references constant nX which is 2, since by construction the redeclared BaseProperties model is in a package with nX = 2.

This definition is compact but is difficult to understand. On first view, an alternative exists, that is more straightforward and is easier to understand:

```

package MoistAir2 "Alternative definition that does not work"
  extends PartialMedium(nX = 2,
    redeclare model BaseProperties = MoistAir_BaseProperties,
    redeclare function dynamicViscosity = MoistAir_dynamicViscosity);

  model MoistAir_BaseProperties // wrong model since nX has no value
    extends PartialMedium.BaseProperties;
  equation
    X = {1,0};
  end MoistAir_BaseProperties;

  model MoistAir_dynamicViscosity
    extends PartialMedium.dynamicViscosity;
  algorithm
    eta :=p;
  end MoistAir_dynamicViscosity;
end MoistAir2;

```

Here, the usual approach is used to extend (here from PartialMedium) and in the modifier perform all redeclarations. In order to perform these redeclarations, corresponding implementations of all elements of PartialMedium have to be given under a different name, such as MoistAir2.MoistAir_BaseProperties, since the name BaseProperties already exists due to “extends PartialMedium”. Then it is possible in the

modifier to redeclare `PartialMedium.BaseProperties` to `MoistAir2.MoistAir_BaseProperties`. Besides the drawback that the namespace is polluted by elements that have different names but the same implementation (e.g. `MoistAir2.BaseProperties` is identical to `MoistAir2.MoistAir_BaseProperties`) the whole construction does not work if arrays are present that depend on constants in `PartialMedium`, such as `X[nX]`: The problem is that `MoistAir_BaseProperties` extends from `PartialMedium.BaseProperties` where the constant `nX` does not yet have a value. This means that the dimension of array `X` is undefined and model `MoistAir_BaseProperties` is wrong. With this construction, all constant definitions have to be repeated whenever these constants shall be used, especially in `MoistAir_BaseProperties` and `MoistAir_dynamicViscosity`. For larger models this is not practical and therefore the only practically useful definition is the complicated construction in the previous example with “**redeclare model extends** `BaseProperties`”.

To detect this issue the rule on lookup of composite names (Section 5.3.2) ensure that ‘`PartialMedium.dynamicViscosity`’ is incorrect in a simulation model.

]

7.3.2 Constraining Type

In a replaceable declaration the optional `constraining_clause` defines a constraining type. Any modifications following the constraining type name are applied both for the purpose of defining the actual constraining type and they are automatically applied in the declaration and in any subsequent redeclaration. If the `constraining_clause` is not present in the original declaration (i.e., the non-redeclared declaration), the type of the declaration is also used as a constraining type and modifications affect the constraining type and are applied in subsequent redeclarations.

The syntax of a `constraining_clause` is as follows:

```
constraining_clause :
    constrainedby name [ class_modification ]
```

[Example:

A modification of the constraining type is automatically applied in subsequent redeclarations:

```
model ElectricalSource
    replaceable SineSource constrainedby MO(final n=5);
    ...
end ElectricalSource;

model TrapezoidalSource
    extends ElectricalSource(
        redeclare Trapezoidal source); // source.n=5
end TrapezoidalSource;
```

A modification of the base type without a constraining type is automatically applied in subsequent redeclarations:

```
model Circuit
    replaceable model NonlinearResistor = Resistor(R=100);
    ...
end Circuit;

model Circuit2
    extends Circuit(
        redeclare replaceable model NonlinearResistor
            = ThermoResistor(T0=300));
        // As a result of the modification on the base type,
        // the default value of R is 100
end Circuit2;

model Circuit3
    extends Circuit2(
        redeclare replaceable model NonlinearResistor
            = Resistor(R=200));
        // The T0 modification is not applied because it did not
```

```
// appear in the original declaration
end Circuit3;
```

A redeclaration can redefine the constraining type:

```
model Circuit4
  extends Circuit2(
    redeclare replaceable model NonlinearResistor
                        = ThermoResistor constrainedby ThermoResistor);
end Circuit4;

model Circuit5
  extends Circuit4(
    redeclare replaceable model NonlinearResistor
                        = Resistor); // illegal
end Circuit5;
]
```

The class or type of component shall be a subtype of the constraining type. In a redeclaration of a replaceable element, the class or type of a component must be a subtype of the constraining type. The constraining type of a replaceable redeclaration must be a subtype of the constraining type of the declaration it redeclares. In an element modification of a replaceable element, the modifications are applied both to the actual type and to the constraining type.

In an element redeclaration of a replaceable element the modifiers of the replaced constraining type are merged to both the new declaration and to the new constraining type, using the normal rules where outer modifiers override inner modifiers.

When a class is flattened as a constraining type, the flattening of its replaceable elements will use the constraining type and not the actual default types.

The number of dimension in the constraining type should correspond to the number of dimensions in the type-part and in the constraining type. Similarly the type used in a redeclaration must have the same number of dimensions as the type of redeclared element.

[Examples:

```
replaceable T1 x[n] constrainedby T2;
replaceable type T=T1[n] constrainedby T2;
replaceable T1[n] x constrainedby T2;
```

In these examples the number of dimensions must be the same in T1 and T2, as well as in a redeclaration. Normally T1 and T2 are scalar types, but both could also be defined as array types – with the same number of dimensions.]

7.3.3 Restrictions on Redeclarations

The following additional constraints apply to redeclarations (after attributes are inherited, Section 7.3):

Only classes and components declared as replaceable can be redeclared with a new type, which must have an interface compatible with the constraining interface of the original declaration, and to allow further redeclarations one must use “`redeclare replaceable`”

- an element declared as constant cannot be redeclared
- an element declared as final may not be modified, and thus not redeclared
- Modelica does not allow a protected element to be redeclared as public, or a public element to be redeclared as protected.
- Array dimensions may be redeclared.

7.3.4 Annotation Choices for Suggested Redeclarations and Modifications

A declaration can have an annotation “choices” containing modifiers on choice, where each of them indicates a suitable redeclaration or modifications of the element.

This is a hint for users of the model, and can also be used by the user interface to suggest reasonable redeclaration, where the string comments on the choice declaration can be used as textual explanations of the choices. The annotation is not restricted to replaceable elements but can also be applied to non-replaceable elements, enumeration types, and simple variables.

[Example:

```
replaceable model MyResistor=Resistor
  annotation(choices(
    choice(redeclare MyResistor=lib2.Resistor(a={2}) "..."),
    choice(redeclare MyResistor=lib2.Resistor2 "...")));

replaceable Resistor Load(R=2) constrainedby TwoPin
  annotation(choices(
    choice(redeclare lib2.Resistor Load(a={2}) "..."),
    choice(redeclare Capacitor Load(L=3) "...")));

replaceable FrictionFunction a(func=exp) constrainedby Friction
  annotation(choices(
    choice(redeclare ConstantFriction a(c=1) "..."),
    choice(redeclare TableFriction a(table=...) "..."),
    choice(redeclare FunctionFriction a(func=exp) "...")));
```

It can also be applied to nonreplaceable declarations, e.g. to describe enumerations.

```
type KindOfController=Integer(min=1,max=3)
  annotation(choices(
    choice=1 "P",
    choice=2 "PI",
    choice=3 "PID"));

model A
  KindOfController x;
end A;
A a(x=3 "PID");
```

]

Chapter 8

Equations

8.1 Equation Categories

Equations in Modelica can be classified into different categories depending on the syntactic context in which they occur:

- *Normal equality equations* occurring in equation sections, including connect-equations and other equation types of special syntactic form (Section 8.3)
- *Declaration equations*, which are part of variable, parameter, or constant declarations (Section 4.4.2.1).
- *Modification equations*, which are commonly used to modify attributes of classes (Section 7.2)
- *Binding equations*, include both declaration equations and modification equations (for the value itself).
- *Initial equations*, which are used to express equations for solving initialization problems (Section 8.6)

8.2 Flattening and Lookup in Equations

A flattened equation is identical to the corresponding nonflattened equation.

Names in an equation shall be found by looking up in the partially flattened enclosing class of the equation.

8.3 Equations in Equation Sections

The following kinds of equations may occur in equation sections. The syntax is defined as follows:

```
equation :
  ( simple_expression "=" expression
    | if_equation
    | for_equation
    | connect_clause
    | when_equation
    | name function_call_args )
comment
```

No statements are allowed in equation sections, including the assignment statement using the `:=` operator.

8.3.1 Simple Equality Equations

Simple equality equations are the traditional kinds of equations known from mathematics that express an equality relation between two expressions. There are two syntactic forms of such equations in Modelica. The first form below is *equality* equations between two expressions, whereas the second form is used when calling a function with *several* results. The syntax for simple equality equations is as follows:

```
simple_expression "=" expression
```

The types of the left-hand-side and the right-hand-side of an equation need to be compatible in the same way as two arguments of binary operators (Section 6.6).

Three examples:

- `simple_expr1 = expr2;`
- `(if pred then alt1 else alt2) = expr2;`
- `(out1, out2, out3) = function_name(inexpr1, inexpr2);`

[Note: According to the grammar the if-then-else expression in the second example needs to be enclosed in parentheses to avoid parsing ambiguities. Also compare with Section 11.2.1.1 about calling functions with several results in assignment statements.]

8.3.2 For-Equations – Repetitive Equation Structures

The syntax of a for-equation is as follows:

```
for for_indices loop
  { equation ";" }
end for ";"
```

For-equations may optionally use several iterators (for_indices), see Section 11.2.2.2 for more information:

```
for_indices:
  for_index {" for_index}

for_index:
  IDENT [ in expression ]
```

The following is one example of a prefix of a for-equation:

```
for IDENT in expression loop
```

The expression of a for-equation shall be a vector expression. It is evaluated once for each for-equation, and is evaluated in the scope immediately enclosing the for-equation. The expression of a for-equation shall be a parameter expression. The loop-variable (IDENT) is in scope inside the loop-construct and shall not be assigned to. The loop-variable has the same type as the type of the elements of the vector expression.

[Example:

```
for i in 1:10 loop           // i takes the values 1,2,3,...,10
for r in 1.0 : 1.5 : 5.5 loop // r takes the values 1.0, 2.5, 4.0, 5.5
for i in {1,3,6,7} loop      // i takes the values 1, 3, 6, 7
for i in TwoEnums loop      // i takes the values TwoEnums.one, TwoEnums.two
                             // for TwoEnums = enumeration(one,two)
```

The loop-variable may hide other variables as in the following example. Using another name for the loop-variable is, however, strongly recommended.

```
constant Integer j=4;
Real x[j];
equation
  for j in 1:j loop // The loop-variable j takes the values 1,2,3,4
    x[j]=j;         // Uses the loop-variable j
  end for;
]
```

8.3.2.1 Implicit Iteration Ranges of For-Equations

The iteration range of a loop variable may sometimes be inferred from its use as an array index. See Section 11.2.2.1 for more information.

[Example:

```
Real x[n], y[n];
```

```

for i loop          // Same as: for i in 1:size(a,1) loop
    x[i] = 2*y[i];
end loop;
]

```

8.3.3 Connect-Equations

A connect-equation has the following syntax:

```
connect "(" component_reference "," component_reference ")" ";"
```

Connect-equations are described in detail in Section 9.1.

8.3.4 If-Equations

If-equations have the following syntax:

```

if expression then
    { equation ";" }
{ elseif expression then
    { equation ";" }
}
[ else
    { equation ";" }
]
end if ";"

```

The expression of an if- or elseif-clause must be a scalar Boolean expression. One if-clause, and zero or more elseif-clauses, and an optional else-clause together form a list of branches. One or zero of the bodies of these if-, elseif- and else-clauses is selected, by evaluating the conditions of the if- and elseif-clauses sequentially until a condition that evaluates to true is found. If none of the conditions evaluate to true the body of the else-clause is selected (if an else-clause exists, otherwise no body is selected). In an equation section, the equations in the body are seen as equations that must be satisfied. The bodies that are not selected have no effect on that model evaluation.

If-equations in equation sections which do not have exclusively parameter expressions as switching conditions shall have an else-clause and each branch shall have the same number of equations. *[If this condition is violated, the single assignment rule would not hold, because the number of equations may change during simulation although the number of unknowns remains the same].*

8.3.5 When-Equations

When-equations have the following syntax:

```

when expression then
    { equation ";" }
{ elseif expression then
    { equation ";" } }
end when ";"

```

The expression of a when-equation shall be a discrete-time Boolean scalar or vector expression. The statements within a when-equation are activated when the scalar expression or any of the elements of the vector expression becomes true.

[Example:

The order between the equations in a when-equation does not matter, e.g.

```

equation
when x > 2 then
    y3 = 2*x + y1 + y2; // Order of y1 and y3 equations does not matter
    y1 = sin(x);

```

```

    end when;
    y2 = sin(y1);
]

```

8.3.5.1 Defining When-Equations by If-Expressions in Equality Equations

A when-equation:

```

equation
  when x>2 then
    v1 = expr1 ;
    v2 = expr2 ;
  end when;

```

is conceptually equivalent to the following equations containing special if-expressions

```

// Not correct Modelica
Boolean b(start=x.start>2);
equation
  b = x>2;
  v1 = if edge(b) then expr1 else pre(v1);
  v2 = if edge(b) then expr2 else pre(v2);

```

[The equivalence is conceptual since `pre()` of a non discrete-time Real variable or expression can only be used within a when-clause. Example:

```

/* discrete*/ Real x;
input Real u;
output Real y;
equation
  when sample() then
    x = a*pre(x)+b*pre(u);
  end when;
y = x;

```

In this example *x* is a discrete-time variable (whether it is declared with the discrete prefix or not), but *u* and *y* cannot be discrete-time variables (since they are not assigned in when-clauses). However, `pre(u)` is legal within the when-clause, since the body of the when-clause is only evaluated at events, and thus all expressions are discrete-time expressions.]

The start-values of the introduced Boolean variables are defined by the taking the start-value of the when-condition, as above where *b* is a parameter variable. The start-values of the special functions `initial`, `terminal`, and `sample` is false.

8.3.5.2 Restrictions on Equations within When-Equations

- When-equations cannot be nested.

[Example:

The following when-equation is invalid:

```

when x > 2 then
  when y1 > 3 then
    y2 = sin(x);
  end when;
end when;
]

```

The equations within the when-equation must have one of the following forms:

- `v = expr;`
- `(out1, out2, out3, ...) = function_call_name(in1, in2, ...);`
- operators `assert()`, `terminate()`, `reinit()`
- For- and if-equations if the equations within the for- and if-equations satisfy these requirements.

- The different branches of when/elsewhen must have the same set of component references on the left-hand side.
- The branches of an if-then-else clause inside when-equations must have the same set of component references on the left-hand side, unless the if-then-else have exclusively parameter expressions as switching conditions.

[The needed restrictions on equations within a when-equation becomes apparent with the following example:

```
Real x, y;
equation
  x + y = 5;
  when condition then
    2*x + y = 7;          // error: not valid Modelica
  end when;
```

When the equations of the when-equation are not activated it is not clear which variable to hold constant, either x or y . A corrected version of this example is:

```
Real x,y;
equation
  x + y = 5;
  when condition then
    y = 7 - 2*x;          // fine
  end when;
```

Here, variable y is held constant when the when-equation is deactivated and x is computed from the first equation using the value of y from the previous event instant.

]

8.3.5.3 Application of the Single-assignment Rule to When-Equations

The Modelica single-assignment rule (Section 8.4) has implications for when-equations:

- Two when-equations may *not* define the same variable.

[Without this rule this may actually happen for the erroneous model `DoubleWhenConflict` below, since there are two equations (`close = true;` `close = false;`) defining the same variable `close`. A conflict between the equations will occur if both conditions would become `true` at the same time instant.

```
model DoubleWhenConflict
  Boolean close;    // Erroneous model: close defined by two equations!
equation
  ...
  when condition1 then
    close = true;
  end when;
  when condition2 then
    close = false;
  end when;
  ...
end DoubleWhenConflict
```

One way to resolve the conflict would be to give one of the two when-equations higher priority. This is possible by rewriting the when-equation using `elsewhen`, as in the `WhenPriority` model below or using the statement version of the when-construct, see Section 11.2.7.]

- When-equations involving `elsewhen`-parts can be used to resolve assignment conflicts since the first of the when/elsewhen parts are given higher priority than later ones:

[Below it is well defined what happens if both conditions become `true` at the same time instant since `condition1` with associated conditional equations has a higher priority than `condition2`.

```
model WhenPriority
  Boolean close;    // Correct model: close defined by two equations!
algorithm
  ...
```

```
    when condition1 then
      close = true;
    elseif condition2 then
      close = false;
    end when;
    ...
  end WhenPriority;
]
```

8.3.6 reinit

The `reinit` operator can be used in the body of a when-equation or a when-statement. It has the following syntax:

```
reinit(x, expr);
```

The operator reinitializes `x` with `expr` at an event instant. `x` is a `Real` variable (or an array of `Real` variables), in which case vectorization applies according to Section 12.4.5) that must be selected as a state (resp., states) at least when the enclosing when clause becomes active. `expr` needs to be type-compatible with `x`. The `reinit` operator can only be applied once for the same variable (resp. array of variables). It can only be applied in the body of a when-equation or when-statement.

The `reinit` operator does not break the single assignment rule, because `reinit(x, expr)` evaluates `expr` to a value (`valu`), then makes the previously known variable `x` unknown and introduces the equation “`x = valu`”.

[If a higher index system is present, i.e., constraints between state variables, some state variables need to be redefined to non-state variables. During simulation, non-state variables should be chosen in such a way that variables with an applied `reinit` operator are selected as states at least when the corresponding when-clauses become active. If this is not possible, an error occurs, since otherwise the `reinit` operator would be applied on a non-state variable.]

Example for the usage of the `reinit` operator:

Bouncing ball:

```
der(h) = v;
der(v) = if flying then -g else 0;
flying = not(h<=0 and v<=0);

when h < 0 then
  reinit(v, -e*pre(v));
end when;
]
```

8.3.7 assert

An equation or statement of the following form:

```
assert(condition, message, level = AssertionLevel.error);
```

is an assertion, where `condition` is a Boolean expression, `message` is a string expression, and `level` is a built-in enumeration with a default value. It can be used in equation sections or algorithm sections.

If the condition of an assertion is true, `message` is not evaluated and the procedure call is ignored. If the condition evaluates to false different actions are taken depending on the level input:

- `level = AssertionLevel.error`: The current evaluation is aborted. The simulation may continue with another evaluation [e.g., with a shorter step-size, or by changing the values of iteration variables]. If the simulation is aborted, `message` indicates the cause of the error.

Failed assertions takes precedence over successful termination, such that if the model first triggers the end of successful analysis by reaching the stop-time or explicitly with `terminate()`, but the evaluation with `terminal()=true` triggers an assert, the analysis failed.

- `level = AssertionLevel.warning`: The current evaluation is not aborted. `message` indicates the cause of the warning [*It is recommended to report the warning only once when the condition becomes false, and it is reported that the condition is no longer violated when the condition returns to true. The `assert(..)` statement shall have no influence on the behavior of the model. For example, by evaluating the condition and reporting the message only after accepted integrator steps. `condition` needs to be implicitly treated with `noEvent(..)` since otherwise events might be triggered that can lead to slightly changed simulation results*].

[The `AssertionLevel.error` case can be used to avoid evaluating a model outside its limits of validity; for instance, a function to compute the saturated liquid temperature cannot be called with a pressure lower than the triple point value.

The `AssertionLevel.warning` case can be used when the boundary of validity is not hard: for instance, a fluid property model based on a polynomial interpolation curve might give accurate results between temperatures of 250 K and 400 K, but still give reasonable results in the range 200 K and 500 K. When the temperature gets out of the smaller interval, but still stays in the largest one, the user should be warned, but the simulation should continue without any further action. The corresponding code would be

```
assert(T > 250 and T < 400, "Medium model outside full accuracy range",
      AssertionLevel.warning);

assert(T > 200 and T < 500, "Medium model outside feasible region");

]
```

8.3.8 terminate

The `terminate(...)` equation or statement [using function syntax] successfully terminates the analysis which was carried out, see also Section 8.3.7. The termination is not immediate at the place where it is defined since not all variable results might be available that are necessary for a successful stop. Instead, the termination actually takes place when the current integrator step is successfully finalized or at an event instant after the event handling has been completed before restarting the integration.

The `terminate` clause has a string argument indicating the reason for the success. [*The intention is to give more complex stopping criteria than a fixed point in time. Example:*

```
model ThrowingBall
  Real x(start=0);
  Real y(start=1);
equation
  der(x)=...
  der(y)=...
algorithm
  when y<0 then
    terminate("The ball touches the ground");
  end when;
end ThrowingBall;

]
```

8.3.9 Equation Operators for Overconstrained Connection-Based Equation Systems

See Section 9.4 for a description of this topic.

8.4 Synchronous Data-flow Principle and Single Assignment Rule

Modelica is based on the synchronous data flow principle and the single assignment rule, which are defined in the following way:

1. All variables keep their actual values until these values are explicitly changed. Variable values can be accessed at any time instant during continuous integration and at event instants.
2. At every time instant, during continuous integration and at event instants, the active equations express relations between variables which have to be fulfilled concurrently (equations are not active if the corresponding if-branch, when-clause or block in which the equation is present is not active).
3. Computation and communication at an event instant does not take time. *[If computation or communication time has to be simulated, this property has to be explicitly modeled].*
4. The total number of equations is identical to the total number of unknown variables (= single assignment rule).

8.5 Events and Synchronization

The integration is halted and an event occurs whenever a Real elementary relation, e.g. “ $x > 2$ ”, changes its value. The value of such a relation can only be changed at event instants *[in other words, Real elementary relations induce state or time events]*. The relation which triggered an event changes its value when evaluated literally before the model is processed at the event instant *[in other words, a root finding mechanism is needed which determines a small time interval in which the relation changes its value; the event occurs at the right side of this interval]*. Relations in the body of a when-clause are always taken literally. During continuous integration a Real elementary relation has the constant value of the relation from the last event instant.

[Example:

```
y = if u > uMax then uMax elseif u < uMin then uMin else u;
```

During continuous integration always the same if-branch is evaluated. The integration is halted whenever u-uMax or u-uMin crosses zero. At the event instant, the correct if-branch is selected and the integration is restarted.

Numerical integration methods of order n ($n \geq 1$) require continuous model equations which are differentiable up to order n . This requirement can be fulfilled if Real elementary relations are not treated literally but as defined above, because discontinuous changes can only occur at event instants and no longer during continuous integration.

]

[It is a quality of implementation issue that the following special relations

```
time >= discrete expression
time < discrete expression
```

trigger a time event at “time = discrete expression”, i.e., the event instant is known in advance and no iteration is needed to find the exact event instant.

]

Relations are taken literally also during continuous integration, if the relation or the expression in which the relation is present, are the argument of the `noEvent(...)` function. The `smooth(p,x)` operator also allows relations used as argument to be taken literally. The `noEvent` feature is propagated to all subrelations in the scope of the `noEvent` function. For `smooth` the liberty to not allow literal evaluation is propagated to all subrelations, but the `smooth`-property itself is not propagated.

[Example:

```
x = if noEvent(u > uMax) then uMax elseif noEvent(u < uMin) then uMin else u;
y = noEvent( if u > uMax then uMax elseif u < uMin then uMin else u );
z = smooth(0, if u > uMax then uMax elseif u < uMin then uMin else u );
```

In this case $x=y=z$, but a tool might generate events for z . The if-expression is taken literally without inducing state events.

The `smooth` function is useful, if e.g. the modeler can guarantee that the used if-clauses fulfill at least the continuity requirement of integrators. In this case the simulation speed is improved, since no state event iterations

occur during integration. The `noEvent` function is used to guard against “outside domain” errors, e.g. `y = if noEvent(x >= 0) then sqrt(x) else 0.`]

All equations and assignment statements within `when`-clauses and all assignment statements within function classes are implicitly treated with the `noEvent` function, i.e., relations within the scope of these operators never induce state or time events. [Using state events in `when`-clauses is unnecessary because the body of a `when`-clause is not evaluated during continuous integration.]

[*Example:*

```
Limit1 = noEvent(x1 > 1);
// Error since Limit1 is a discrete-time variable

when noEvent(x1>1) or x2>10 then
  // error, when-conditions is not a discrete-time expression
  Close = true;
end when;
]
```

Modelica is based on the synchronous data flow principle (Section 8.2).

[The rules for the synchronous data flow principle guarantee that variables are always defined by a unique set of equations. It is not possible that a variable is e.g. defined by two equations, which would give rise to conflicts or non-deterministic behavior. Furthermore, the continuous and the discrete parts of a model are always automatically “synchronized”. Example:

```
equation // Illegal example
  when condition1 then
    close = true;
  end when;

  when condition2 then
    close = false;
  end when;
```

This is not a valid model because rule 4 is violated since there are two equations for the single unknown variable `close`. If this would be a valid model, a conflict occurs when both conditions become true at the same time instant, since no priorities between the two equations are assigned. To become valid, the model has to be changed to:

```
equation
  when condition1 then
    close = true;
  elseif condition2 then
    close = false;
  end when;
```

Here, it is well-defined if both conditions become true at the same time instant (`condition1` has a higher priority than `condition2`).

]

There is no guarantee that two different events occur at the same time instant.

[As a consequence, synchronization of events has to be explicitly programmed in the model, e.g. via counters. Example:

```
Boolean fastSample, slowSample;
Integer ticks(start=0);
equation
  fastSample = sample(0,1);
algorithm
  when fastSample then
    ticks := if pre(ticks) < 5 then pre(ticks)+1 else 0;
    slowSample := pre(ticks) == 0;
  end when;
algorithm
  when fastSample then // fast sampling
    ...
```

```
    end when;  
algorithm  
  when slowSample then    // slow sampling (5-times slower)  
    ...  
  end when;
```

The `slowSample` when-clause is evaluated at every 5th occurrence of the `fastSample` when-clause.

]

[The single assignment rule and the requirement to explicitly program the synchronization of events allow a certain degree of model verification already at compile time.]

8.6 Initialization, initial equation, and initial algorithm

Before any operation is carried out with a Modelica model [e.g., *simulation or linearization*], initialization takes place to assign consistent values for all variables present in the model. During this phase, also the derivatives, `der(..)`, and the pre-variables, `pre(..)`, are interpreted as unknown algebraic variables. The initialization uses all equations and algorithms that are utilized in the intended operation [such as *simulation or linearization*]. The equations of a when-clause are active during initialization, if and only if they are explicitly enabled with the “`initial()`” operator. In this case, the when-clause equations remain active during the whole initialization phase. [If a when-clause equation $v = \text{expr};$ is not active during the initialization phase, the equation $v = \text{pre}(v)$ is added for initialization. This follows from the mapping rule of when-clause equations].

Further constraints, necessary to determine the initial values of all variables, can be defined in the following two ways:

1) As equations in an `initial equation` section or as assignments in an `initial algorithm` section. The equations and assignments in these initial sections are purely algebraic, stating constraints between the variables at the initial time instant. It is not allowed to use when-clauses in these sections.

2) Implicitly by using the attributes `start=value` and `fixed=true` in the declaration of variables:

- For all non-discrete Real variables v , the equation “ $v = \text{startExpression}$ ” is added to the initialization equations, if “`start = startExpression`” and “`fixed = true`”.
- For all discrete variables vd , the equation “`pre(vd) = startExpression`” is added to the initialization equations, if “`start = startExpression`” and “`fixed = true`”.
- For constants and parameters, the attribute `fixed` is by default `true`. For other variables `fixed` is by default `false`.

[A Modelica translator may first transform the continuous equations of a model, at least conceptually, to state space form. This may require to differentiate equations for index reduction, i.e., additional equations and, in some cases, additional unknown variables are introduced. This whole set of equations, together with the additional constraints defined above, should lead to an algebraic system of equations where the number of equations and the number of all variables (including `der(..)` and `pre(..)` variables) is equal. Often, this is a nonlinear system of equations and therefore it may be necessary to provide appropriate guess values (i.e., `start` values and `fixed=false`) in order to compute a solution numerically.

It may be difficult for a user to figure out how many initial equations have to be added, especially if the system has a higher index. A tool may add or remove initial equations automatically such that the resulting system is structurally nonsingular. In these cases diagnostics are appropriate since the result is not unique and may not be what the user expects. A missing initial value of a discrete variable which does not influence the simulation result, may be automatically set to the start value or its default without informing the user. For example, variables assigned in a when-clause which are not accessed outside of the when-clause and where the `pre()` operator is not explicitly used on these variables, do not have an effect on the simulation.

Examples:

Continuous time controller initialized in steady-state:

```

Real y(fixed = false); // fixed=false is redundant
equation
  der(y) = a*y + b*u;
initial equation
  der(y) = 0;

```

This has the following solution at initialization:

```

der(y) = 0;
y = -b/a *u;

```

Continuous time controller initialized either in steady-state or by providing a start value for state y:

```

parameter Boolean steadyState = true;
parameter Real y0 = 0 "start value for y, if not steadyState";
Real y;
equation
  der(y) = a*y + b*u;
initial equation
  if steadyState then
    der(y)=0;
  else
    y = y0;
  end if;

```

This can also be written as follows (this form is less clear):

```

parameter Boolean steadyState=true;
Real y (start=0, fixed=not steadyState);
Real der_y(start=0, fixed=steadyState) = der(y);
equation
  der(y) = a*y + b*u;

```

Discrete time controller initialized in steady-state:

```

discrete Real y;
equation
  when {initial(), sampleTrigger} then
    y = a*pre(y) + b*u;
  end when;
initial equation
  y = pre(y);

```

This leads to the following equations during initialization:

```

y = a*pre(y) + b*u;
y = pre(y);

```

With the solution:

```

y := (b*u)/(1-a)
pre(y) := y;

```

]

8.6.1 The Number of Equations Needed for Initialization

[In general, for the case of a pure ordinary differential equation (ODE) system with n state variables and m output variables, we will have $n+m$ unknowns in the simulation problem. The ODE initialization problem has n additional unknowns corresponding to the derivative variables. At initialization of an ODE we will need to find the values of $2n+m$ variables, in contrast to just $n+m$ variables to be solved for during simulation.]

Example: Consider the following simple equation system:

```

der(x1) = f1(x1);
der(x2) = f2(x2);
y = x1+x2+u;

```

Here we have three variables with unknown values: two dynamic variables that also are state variables, x_1 and x_2 , i.e., $n=2$, one output variable y , i.e., $m=1$, and one input variable u with known value. A consistent solution of the initial value problem providing initial values for x_1 , x_2 , $\text{der}(x_1)$, $\text{der}(x_2)$, and y needs to be found. Two additional initial equations thus need to be provided to solve the initialization problem.

Regarding DAEs, only that at most n additional equations are needed to arrive at $2n+m$ equations in the initialization system. The reason is that in a higher index DAE problem the number of dynamic continuous-time state variables might be less than the number of state variables n . As noted in Section 8.6 a tool may add/remove initial equations to fulfill this requirement, if appropriate diagnostics are given.

]

Chapter 9

Connectors and Connections

This chapter covers connectors, connect-equations, and connections.

The connect-equations (and the special functions for overdetermined connectors) may only be used in equations and may not be used inside if-equations with non-parametric condition, or in when-equations. [For-equations always have parameter expressions for the array expression.]

The special function cardinality [*to be deprecated*] may not be used to control them.

9.1 Connect-Equations and Connectors

Connections between objects are introduced by connect-equations in the equation part of a class. A connect-equation has the following syntax:

```
connect "(" component_reference "," component_reference ")" ";"
```

The connect-equation construct takes two references to connectors [*a connector is an instance of a connector class*], each of which is either of the following forms:

- $c_1, c_2 \dots c_n$, where c_1 is a connector of the class, $n \geq 1$ and c_{i+1} is a connector element of c_i for $i = 1 : (n-1)$.
- $m.c$, where m is a non-connector element in the class and c is a connector element of m .

There may optionally be array subscripts on any of the components; the array subscripts shall be parameter expressions. If the connect construct references array of connectors, the array dimensions must match, and each corresponding pair of elements from the arrays is connected as a pair of scalar connectors.

[*Example of array usage:*

```
connector InPort=input Real;
connector OutPort=output Real;

block MatrixGain
  input InPort u[size(A,1)];
  output OutPort y[size(A,2)]
  parameter Real A[:,:]=[1];
equation
  y=A*u;
end MatrixGain;

sin sinSource[5];
MatrixGain gain(A=5*identity(5));
MatrixGain gain2(A=ones(5,2));
OutPort x[2];
equation
connect(sinSource.y, gain.u); // Legal
connect(gain.y, gain2.u);     // Legal
connect(gain2.y, x);         // Legal
```

]

The three main tasks are to:

- Elaborate expandable connectors.
- Build connection sets from connect-equations.
- Generate equations for the complete model.

9.1.1 Connection Sets

A connection set is a set of variables connected by means of connect-equations. A connection set shall contain either only flow variables or only non-flow variables.

9.1.2 Inside and Outside Connectors

In an element instance *M*, each connector element of *M* is called an outside connector with respect to *M*. All other connector elements that are hierarchically inside *M*, but not in one of the outside connectors of *M*, is called an inside connector with respect to *M*.

[*Example: in connect(a,b.c) a is an outside connector and b.c is an inside connector, unless b is a connector.*]

9.1.3 Expandable Connectors

If the `expandable` qualifier is present on a connector definition, all instances of that connector are referred to as expandable connectors. Instances of connectors that do not possess this qualifier will be referred to as non-expandable connectors.

Before generating connection equations, connections containing expandable connectors are elaborated:

- When two expandable connectors are connected, each is augmented with the variables that are only declared in the other expandable connector (the new variables are neither input nor output). This is repeated until all connected expandable connector instances have matching variables [*i.e. each of the connector instances is expanded to be the union of all connector variables.*]

The following rules apply to expandable connectors

- expandable connectors can only be connected to other expandable connectors.
- The variables introduced in the elaboration follow additional rules for default-connection (given in Section 9.2).
- If a variable appears as an input in one expandable connector, it should appear as a non-input in at least one other expandable connector instance in the same augmentation set.

[*Example:*

```
expandable connector EngineBus
end EngineBus;

block Sensor
  RealOutput speed;
end Sensor;

block Actuator
  RealInput speed;
end Actuator;

model Engine
  EngineBus bus;
  Sensor sensor;
  Actuator actuator;
```

```

equation
  connect(bus.speed, sensor.speed); // provides the non-input
  connect(bus.speed, actuator.speed);
end Engine;
]

```

- All components in an expandable connector are seen as connector instances even if they are not declared as such [*i.e. it is possible to connect to e.g. a Real variable*].

[Example:

```

expandable connector EngineBus // has predefined signals
  import SI=Modelica.SIunits;
  SI.AngularVelocity speed;
  SI.Temperature T;
end EngineBus;

block Sensor
  RealOutput speed;
end Sensor;

model Engine
  EngineBus bus;
  Sensor sensor;
equation
  connect(bus.speed, sensor.speed);
  // connection to non-connector speed is possible
  // in expandable connectors
end Engine;
]

```

- An expandable connector may not contain a component declared with the prefix `flow`, but may contain non-expandable connector components with `flow` components.

[Example:

```

import Interfaces=Modelica.Electrical.Analog.Interfaces;

expandable connector ElectricalBus
  Interfaces.PositivePin p12, n12; // OK
  flow Modelica.SIunits.Current i; // not allowed
end ElectricalBus;

model Battery
  Interfaces.PositivePin p42, n42;
  ElectricalBus bus;
equation
  connect(p42, bus.p42); // Adds new electrical pin
  connect(n42, bus.n42); // Adds another pin
end Battery;
]

```

- One connector in the connect equation must reference a declared component and the (other) undeclared component may not be subscripted:
 - The expandable connector instance is automatically augmented with a new component having the used name and corresponding type.
 - If the variable on the other side of the connect-equation is input or output the new component will be either input or output to satisfy the restrictions in Section 9.3. [*If the existing side refers to an inside connector (i.e. a connector of a component) the new variable will copy its causality, i.e. input if input and output if output, since the expandable connector must be an outside connector*].

[Example:

```

expandable connector EmptyBus
end EmptyBus;

model Controller

```

```
EmptyBus bus1;
EmptyBus bus2;
RealInput speed;
equation
  connect(speed, bus1.speed); // ok, only one undeclared
  // and it is unsubscripted
  connect(bus1.pressure, bus2.pressure);
  // not allowed, both undeclared
  connect(speed, bus2.speed[2]);
  // not allowed, undeclared is subscripted
end Controller;
]
```

After this elaboration the expandable connectors are treated as normal connector instances, and the connections as normal connections. This elaboration implies that expandable connectors can be connected even if they do not contain the same components.

[Note that the introduction of variables, as described above, is conceptual and does not necessarily impact the flattening hierarchy in any way. Furthermore, it is important to note that these elaboration rules must consider:

- 1) Expandable connectors nested hierarchically. This means that both outside and inside connectors must be included at every level of the hierarchy in this elaboration process.*
- 2) When processing an expandable connector that possesses the inner scope qualifier, all outer instances must also be taken into account during elaboration.*

Example:

Engine system with sensors, controllers, actuator and plant that exchange information via a bus (i.e. via expandable connectors):

```
import SI=Modelica.SIunits;
import Modelica.Blocks.Interfaces.*;

// Plant Side
model SparkPlug
  RealInput spark_advance spark_advance;
  ...
end SparkPlug;

expandable connector EngineBus
  // No minimal set
end EngineBus;

expandable connector CylinderBus
  RealInput spark_advance; // minimal set
end CylinderBus;

model Cylinder
  CylinderBus cylinder_bus;
  SparkPlug spark_plug;
  ...
equation
  // doesn't need to expand bus since spark_advance is in minimal set
  connect(spark_plug.spark_advance, cylinder_bus.spark_advance);
end Cylinder;

model I4 "Non array implementation"
  EngineBus engine_bus;
  Modelica.Mechanics.Rotational.Sensors.SpeedSensor speed_sensor;
  Modelica.Thermal.Sensors.TemperatureSensor temp_sensor;
  Cylinder cylinder1;
  Cylinder cylinder2;
  Cylinder cylinder3;
  Cylinder cylinder4;
equation
  // adds engine_speed (as output)
  connect(speed_sensor.w, engine_bus.engine_speed);
  // adds engine_temp (as output)
```



```

connect(temp_sensor.T, engine_bus.engine_temp);
// adds cylinder_bus1 (a nested bus)
connect(cylinder1.cylinder_bus, engine_bus.cylinder_bus1);
// adds cylinder_bus2 and cylinder_bus3
...
// adds cylinder_bus4 (a nested bus)
connect(cylinder4.cylinder_bus, engine_bus.cylinder_bus4);
// Not allowed: connecting to a sub-scripted undeclared component:
// connect(cylinder1.cylinder_bus, engine_bus.cylinder_bus[1]);
//
// Not allowed: introducing several levels by connection:
// (since cylinder_bus1 is not declared).
// connect(temp_sensor.T, engine_bus.cylinder_bus1.eng_temp);
end I4;

```

Due to the above connection, conceptually a connector consisting of the union of all connectors is introduced.

The engine_bus contains the following variable declarations:

```

RealOutput engine_speed;
RealOutput engine_temp;
CylinderBus cylinder_bus1;
CylinderBus cylinder_bus2;
CylinderBus cylinder_bus3;
CylinderBus cylinder_bus4;

model I4_array "Array implementation"
  EngineBus engine_bus;
  Modelica.Mechanics.Rotational.Sensors.SpeedSensor speed_sensor;
  Modelica.Thermal.Sensors.TemperatureSensor temp_sensor;
  Cylinder cylinder[4];
equation
  // adds engine_speed (as output)
  connect(speed_sensor.w, engine_bus.engine_speed);
  // adds engine_temp (as output)
  connect(temp_sensor.T, engine_bus.engine_temp);
  // with slicing, adds cylinder_bus[4] (an array of buses)
  connect(cylinder.cylinder_bus, engine_bus.cylinder_bus);
end I4_array;

```

Due to the above connection, conceptually a connector consisting of the union of all connectors is introduced.

The engine_bus contains the following variable declarations:

```

RealOutput engine_speed;
RealOutput engine_temp;
CylinderBus cylinder_bus[4];
]

```

9.2 Generation of Connection Equations

Before generating connection equations outer elements are resolved to the corresponding inner elements in the instance hierarchy (see instance hierarchy name lookup 3.1.1.3). The arguments to each connect-equation are resolved to two connector elements, and the connection is moved up zero or more times in the instance hierarchy to the first element instance that both the connectors are hierarchically contained in it.

For every use of the connect-equation

```
connect(a, b);
```

the primitive components of a and b form a connection set. If any of them already occur in a connection set from previous connections with matching inside/outside, these sets are merged to form one connection set. Composite connector types are broken down into primitive components. Each connection set is used to generate equations for across and through (zero-sum) variables of the form

- $a_1 = a_2 = \dots = a_n$;

- $z_1 + z_2 + (-z_3) + \dots + z_n = 0$;

In order to generate equations for through variables [*using the flow prefix*], the sign used for the connector variable z_1 above is +1 for inside connectors and -1 for outside connectors [z_3 in the example above].

For each flow (zero-sum) variable in a connector that is not connected as an inside connector in any element instance (or if the variable was introduced during augmentation of expandable connectors and the expandable connector is not connected as an outside connector or if the connector is declared as an inner component and not connected as an outside connector) the following equation is implicitly generated:

$$\mathbf{z} = \mathbf{0};$$

The bold-face **0** represents an array or scalar zero of appropriate dimensions (i.e. the same size as \mathbf{z}).

9.3 Restrictions of Connections and Connectors

- In a connect-equation the primitive components of the two connectors must have the same primitive types, and flow-variables may only connect to other flow-variables, causal variables (input/output) only to causal variables (input/output).
- A connection set of causal variables (input/output) may at most contain one inside output connector or one public outside input connector. [*i.e., a connection set may at most contain one source of a signal.*]
- At least one of the following must hold for a connection set containing causal variables: (1) the model or block is partial, (2) the connection set includes variables from an outside public expandable connector, (3) the set contains protected outside connectors, (4) it contains one inside output connector, or (5) one public outside input connector, or (6) the set is comprised solely of an inside input connector that is not part of an expandable connector. [*i.e., a connection set must – unless the model or block is partial – contain one source of a signal (the last item covers the case where connector of the block is left unconnected and the source given textually).*]
- A protected outside connector must be part of a connection set containing at least one inside connector or one declared public outside connector (i.e. it may not be an implicitly defined part of an expandable connector). [*Otherwise it would not be possible to deduce the causality for the expandable connector element.*]
- A connect statement may not (directly or indirectly) connect two connectors of outer elements. [*indirectly is similar to them being part of the same connection set – however, connections to outer elements are “moved up” before forming connection sets. Otherwise the connection sets could contain “redundant” information breaking the equation count for locally balanced models and blocks.*]
- Subscripts in a connector reference shall be parameter expressions or the special operator “.”.
- If the array sizes do not match, the original variables are filled with one-sized dimensions from the left until the number of dimensions match before the connection set equations are generated.
- Constants or parameters in connected components yield the appropriate assert statements; connections are not generated.

9.3.1 Size Restriction on Connectors

For each non-partial connector class the number of flow variables shall be equal to the number of variables that are neither parameter, constant, input, output nor flow. The “number of variables” is the number of all elements in the connector class after expanding all records and arrays to a set of scalars of primitive types. The number of variables of an overdetermined type or record class (see Section 9.4.1) is the size of the output argument of the corresponding equalityConstraint() function.

[Examples:

```
connector Pin // a physical connector of Modelica.Electrical.Analog
  Real v;
  flow Real i;
```

```

end Pin;

connector Plug // a hierachical connector of Modelica.Electrical.MultiPhase
  parameter Integer m=3;
  Pin p[m];
end Plug;

connector InputReal = input Real; // A causal input connector

connector OutputReal = output Real; // A causal output connector

connector Frame_Illegal
  Modelica.SIunits.Position      r0[3]  "Position vector of frame origin";
  Real                          S[3, 3] "Rotation matrix of frame";
  Modelica.SIunits.Velocity      v[3]   "Abs. velocity of frame origin";
  Modelica.SIunits.AngularVelocity w[3]  "Abs. angular velocity of frame";
  Modelica.SIunits.Acceleration  a[3]   "Abs. acc. of frame origin";
  Modelica.SIunits.AngularAcceleration z[3] "Abs. angular acc. of frame";
  flow Modelica.SIunits.Force    f[3]   "Cut force";
  flow Modelica.SIunits.Torque   t[3]   "Cut torque";
end Frame_Illegal;

```

The `Frame_Illegal` connector (intended to be used in a simple `MultiBody`-package without over-determined connectors) is illegal since the number of flow and non-flow variables do not match. The solution is to create two connectors classes, where two 3-vectors (e.g., a and z) are acausal `Real` and the other variables are matching pairs of input and output. This ensures that the models can only be connected in a tree-structure or require a “loop-breaker” joint for every closed kinematic loop:

```

connector Frame_a "correct connector"
  input Modelica.SIunits.Position      r0[3];
  input Real                          S[3, 3];
  input Modelica.SIunits.Velocity      v[3];
  input Modelica.SIunits.AngularVelocity w[3];
  Modelica.SIunits.Acceleration        a[3];
  Modelica.SIunits.AngularAcceleration z[3];
  flow Modelica.SIunits.Force          f[3];
  flow Modelica.SIunits.Torque         t[3];
end Frame_a;

connector Frame_b "correct connector"
  output Modelica.SIunits.Position      r0[3];
  output Real                          S[3, 3];
  output Modelica.SIunits.Velocity      v[3];
  output Modelica.SIunits.AngularVelocity w[3];
  Modelica.SIunits.Acceleration        a[3];
  Modelica.SIunits.AngularAcceleration z[3];
  flow Modelica.SIunits.Force          f[3];
  flow Modelica.SIunits.Torque         t[3];
end Frame_b;

```

The subsequent connectors `Plug_Expanded` and `PlugExpanded2` are correct, but `Plug_Expanded_Illegal` is illegal since the number of non-flow and flow variables is different if “ n ” and “ m ” are different. It is not clear how a tool can detect in general that connectors such as `Plug_Expanded_Illegal` are illegal. However, it is always possible to detect this defect after actual values of parameters and constants are provided in the simulation model.

```

connector Plug_Expanded "correct connector"
  parameter Integer m=3;
  Real          v[m];
  flow Real i[m];
end Plug_Expanded;

connector Plug_Expanded2 "correct connector"
  parameter Integer m=3;
  final parameter Integer n=m;
  Real          v[m];

```

```

    flow Real i[n];
  end Plug_Expanded2;

  connector Plug_Expanded_Illegal "connector is illegal"
    parameter Integer m=3;
    parameter Integer n=m;
    Real v[m];
    flow Real i[n];
  end Plug_Expanded_Illegal;
]

```

9.4 Equation Operators for Overconstrained Connection-Based Equation Systems

There is a special problem regarding equation systems resulting from *loops* in connection graphs where the connectors contain *nonflow* (i.e., potential) variables *dependent* on each other. When a loop structure occurs in such a graph, the resulting equation system will be *overconstrained*, i.e., have more equations than variables, since there are implicit constraints between certain nonflow variables in the connector in addition to the connection equations around the loop. At the current state-of-the-art, it is not possible to automatically eliminate the unneeded equations from the resulting equation system without additional information from the model designer.

This section describes a set of equation operators for such overconstrained connection-based equation systems, that makes it possible for the model designer to specify enough information in the model to allow a Modelica environment to automatically remove the superfluous equations.

[Connectors may contain redundant variables. For example, the orientation between two coordinate systems in 3 dimensions can be described by 3 independent variables. However, every description of orientation with 3 variables has at least one singularity in the region where the variables are defined. It is therefore not possible to declare only 3 variables in a connector. Instead n variables ($n > 3$) have to be used. These variables are no longer independent from each other and there are $n-3$ constraint equations that have to be fulfilled. A proper description of a redundant set of variables with constraint equations does no longer have a singularity. A model that has loops in the connection structure formed by components and connectors with redundant variables, may lead to a differential algebraic equation system that has more equations than unknown variables. The superfluous equations are usually consistent with the rest of the equations, i.e., a unique mathematical solution exists. Such models cannot be treated with the currently known symbolic transformation methods. To overcome this situation, operators are defined in order that a Modelica translator can remove the superfluous equations. This is performed by replacing the equality equations of non-flow variables from connection sets by a reduced number of equations in certain situations.]

This section handles a certain class of overdetermined systems due to connectors that have a redundant set of variables. There are other causes of overdetermined systems, e.g., explicit zero-sum equations for flow variables, that are not handled by the method described below.]

9.4.1 Overconstrained Equation Operators for Connection Graphs

A type or record declaration may have an optional definition of function “equalityConstraint(..)” that shall have the following prototype:

```

type Type // overdetermined type
  extends <base type>;

function equalityConstraint // non-redundant equality
  input Type T1;
  input Type T2;
  output Real residue[ <n> ];
algorithm
  residue := ...

```

```

    end equalityConstraint;
end Type;

record Record
  < declaration of record fields>

  function equalityConstraint // non-redundant equality
    input  Record R1;
    input  Record R2;
    output Real residue[ <n> ];
  algorithm
    residue := ...
  end equalityConstraint;
end Record;

```

The “residue” output of the `equalityConstraint(..)` function shall have known size, say constant n . The function shall express the equality between the two type instances $T1$ and $T2$ or the record instances $R1$ and $R2$, respectively, with a non-redundant number $n \geq 0$ of equations. The residues of these equations are returned in vector “residue” of size n . The set of n non-redundant equations stating that $R1 = R2$ is given by the equation (0 characterizes a vector of zeros of appropriate size):

```

Record R1, R2;
equation
  0 = Record.equalityConstraint(R1,R2);

```

[If the elements of a record `Record` are not independent from each other, the equation “ $R1 = R2$ ” contains redundant equations].

A type class with an `equalityConstraint` function declaration is called overdetermined type. A record class with an `equalityConstraint` function definition is called overdetermined record. A connector that contains instances of overdetermined type and/or record classes is called overdetermined connector. An overdetermined type or record may neither have flow components nor may be used as a type of flow components.

Every instance of an overdetermined type or record in an overdetermined connector is a node in a virtual connection graph that is used to determine when the standard equation “ $R1 = R2$ ” or when the equation “ $0 = \text{equalityConstraint}(R1,R2)$ ” has to be used for the generation of `connect(...)` equations. The branches of the virtual connection graph are implicitly defined by “`connect(..)`” and explicitly by `Connections.branch(...)` statements, see table below. `Connections` is a built-in package in global scope containing built-in operators. Additionally, corresponding nodes of the virtual connection graph have to be defined as roots or as potential roots with functions `Connections.root(...)` and `Connections.potentialRoot(...)`, respectively. In the following table, A and B are connector instances that may be hierarchically structured, e.g., A may be an abbreviation for `EnginePort.Frame`.

<code>connect(A,B);</code>	Defines <i>breakable branches</i> from the overdetermined type or record instances in connector instance A to the corresponding overdetermined type or record instances in connector instance B for a virtual connection graph. The types of the corresponding overdetermined type or record instances shall be the same.
<code>Connections.branch(A.R,B.R);</code>	Defines a <i>non-breakable branch</i> from the overdetermined type or record instance R in connector instance A to the corresponding overdetermined type or record instance R in connector instance B for a virtual connection graph. This function can be used at all places where a <code>connect(..)</code> statement is allowed [e.g., it is not allowed to use this function in a <i>when-clause</i> . This definition shall be used if in a model with connectors A and B the overdetermined records $A.R$ and $B.R$ are algebraically coupled in the model, e.g., due to $B.R = f(A.R, \text{<other unknowns>})$].

<code>Connections.root(A.R);</code>	The overdetermined type or record instance R in connector instance A is a (definite) <i>root node</i> in a virtual connection graph. <i>[This definition shall be used if in a model with connector A the overdetermined record A.R is (consistently) assigned, e.g., from a parameter expressions]</i>
<code>Connections.potentialRoot(A.R);</code> <code>Connections.potentialRoot(A.R, priority = p);</code>	The overdetermined type or record instance R in connector instance A is a <i>potential root node</i> in a virtual connection graph with priority “p” ($p \geq 0$). If no second argument is provided, the priority is zero. “p” shall be a parameter expression of type Integer. In a virtual connection subgraph without a <code>Connections.root</code> definition, one of the potential roots with the lowest priority number is selected as root <i>[This definition may be used if in a model with connector A the overdetermined record A.R appears differentiated – <code>der(A.R)</code> – together with the constraint equations of A.R, i.e., a non-redundant subset of A.R maybe used as states]</i>
<code>b = Connections.isRoot(A.R);</code>	Returns true, if the overdetermined type or record instance R in connector instance A is selected as a root in the virtual connection graph.

[Note, that `Connections.branch`, `Connections.root`, `Connections.potentialRoot` do not generate equations. They only generate nodes and branches in the virtual graph for analysis purposes.]

9.4.2 Converting the Connection Graph into Trees and Generating Connection Equations

Before `connect(...)` equations are generated, the virtual connection graph is transformed into a set of spanning trees by removing breakable branches from the graph. This is performed in the following way:

1. Every root node defined via the “`Connections.root(...)`” statement is a definite root of one spanning tree.
2. The virtual connection graph may consist of sets of subgraphs that are not connected together. Every subgraph in this set shall have at least one root node or one potential root node in a simulation model. If a graph of this set does not contain any root node, then one potential root node in this subgraph that has the lowest priority number is selected to be the root of that subgraph. The selection can be inquired in a class with function `Connections.isRoot(...)`, see table above.
3. If there are n selected roots in a subgraph, then breakable branches have to be removed such that the result shall be a set of n spanning trees with the selected root nodes as roots.

After this analysis, the connection equations are generated in the following way:

1. For every breakable branch [*i.e., a `connect(A,B)` equation,*] in one of the spanning trees, the connection equations are generated according to Section 9.2.
2. For every breakable branch not in any of the spanning trees, the connection equations are generated according to Section 9.2, except for overdetermined type or record instances R. Here the equations “`0 = R.equalityConstraint(A.R,B.R)`” are generated instead of “`A.R = B.R`”.

9.4.3 Examples of Overconstrained Connection Graphs

[Example:

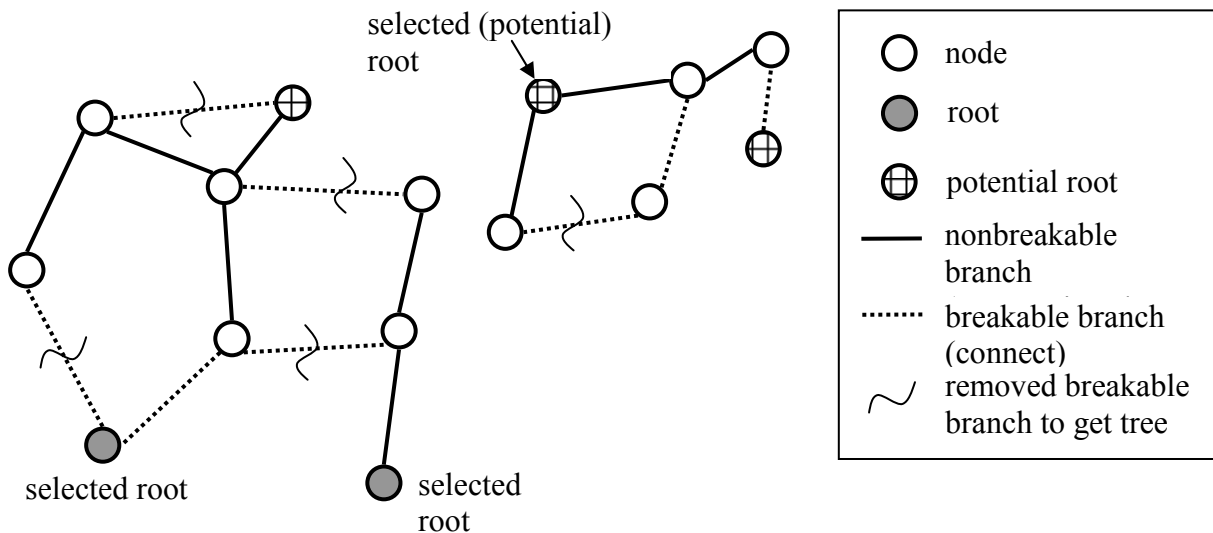


Figure 9-1. Example of a virtual connection graph.

]

9.4.3.1 An Overdetermined Connector for Power Systems

[An overdetermined connector for power systems based on the transformation theory of Park may be defined as:

```

type AC_Angle "Angle of source, e.g., rotor of generator"
  extends Modelica.SIunits.Angle; // AC_Angle is a Real number
                                   // with unit = "rad"

  function equalityConstraint
    input AC_Angle theta1;
    input AC_Angle theta2;
    output Real residue[0] "No constraints"
  algorithm
    /* make sure that theta1 and theta2 from
       joining branches are identical */
    assert(abs(theta1 - theta2) < 1.e-10);
  end equalityConstraint;
end AC_Angle;

connector AC_Plug "3-phase alternating current connector"
  import SI = Modelica.SIunits;
  AC_Angle theta;
  SI.Voltage v[3] "Voltages resolved in AC_Angle frame";
  flow SI.Current i[3] "Currents resolved in AC_Angle frame";
end AC_Plug;

```

The currents and voltages in the connector are defined relatively to the harmonic, high-frequency signal of a power source that is essentially described by angle θ of the rotor of the source. This allows much faster simulations, since the basic high frequency signal of the power source is not part of the differential equations. For example, when the source and the rest of the line operates with constant frequency (= nominal case), then $AC_Plug.v$ and $AC_Plug.i$ are constant. In this case a variable step integrator can select large time steps. An element, such as a 3-phase inductor, may be implemented as:

```

model AC_Inductor
  parameter Real X[3,3], Y[3,3]; // component constants
  AC_Plug p;
  AC_Plug n;
equation
  Connections.branch(p.theta,n.theta); //branch in virtual graph
                                       // since n.theta = p.theta
  n.theta = p.theta;                  // pass angle theta between plugs
  omega = der(p.theta);                // frequency of source
  zeros(3) = p.i + n.i;
end AC_Inductor;

```

```

X*der(p.i) + omega*Y*p.i = p.v - n.v;
end AC_Inductor

```

At the place where the source frequency, i.e., essentially variable *theta*, is defined, a `Connections.root(...)` must be present:

```

AC_plug p;
equation
  Connections.root(p.theta);
  der(p.theta) = 2*Modelica.Constants.pi*50 // 50 Hz;

```

The graph analysis performed with the virtual connection graph identifies the connectors, where the `AC_Angle` needs not to be passed between components, in order to avoid redundant equations.

9.4.3.2 An Overdetermined Connector for 3-dimensional Mechanical Systems

An overdetermined connector for 3-dimensional mechanical systems may be defined as:

```

type TransformationMatrix = Real[3,3];

type Orientation "Orientation from frame 1 to frame 2"
  extends Real[3,3];

function equalityConstraint
  input Orientation R1 "Rotation from inertial frame to frame 1";
  input Orientation R2 "Rotation from inertial frame to frame 2";
  output Real residue[3];
protected
  Orientation R_rel "Relative Rotation from frame 1 to frame 2";
algorithm
  R_rel = R2*transpose(R1);
  /* If frame_1 and frame_2 are identical, R_rel must be
     the unit matrix. If they are close together, R_rel can be
     linearized yielding:
     R_rel = [ 1,   phi3, -phi2;
              -phi3, 1,   phi1;
              phi2, -phi1, 1 ];
     where phi1, phi2, phi3 are the small rotation angles around
     axis x, y, z of frame 1 to rotate frame 1 into frame 2.
     The atan2 is used to handle large rotation angles, but does not
     modify the result for small angles.
  */
  residue := { Modelica.Math.atan2(R_rel[2, 3], R_rel[1, 1]),
              Modelica.Math.atan2(R_rel[3, 1], R_rel[2, 2]),
              Modelica.Math.atan2(R_rel[1, 2], R_rel[3, 3]) };
end equalityConstraint;
end Orientation;

connector Frame "3-dimensional mechanical connector"
  import SI = Modelica.SIunits;
  SI.Position r[3] "Vector from inertial frame to Frame";
  Orientation R "Orientation from inertial frame to Frame";
  flow SI.Force f[3] "Cut-force resolved in Frame";
  flow SI.Torque t[3] "Cut-torque resolved in Frame";
end Frame;

```

A fixed translation from a frame A to a frame B may be defined as:

```

model FixedTranslation
  parameter Modelica.SIunits.Position r[3];
  Frame frame_a, frame_b;
equation
  Connections.branch(frame_a.R, frame_b.R);
  frame_b.r = frame_a.r + transpose(frame_a.R)*r;
  frame_b.R = frame_a.R;
  zeros(3) = frame_a.f + frame_b.f;
  zeros(3) = frame_a.t + frame_b.t + cross(r, frame_b.f);
end FixedTranslation;

```


Since the transformation matrix `frame_a.R` is algebraically coupled with `frame_b.R`, a branch in the virtual connection graph has to be defined. At the inertial system, the orientation is consistently initialized and therefore the orientation in the inertial system connector has to be defined as root:

```
model InertialSystem
  Frame frame_b;
equation
  Connections.root(frame_b.R);
  frame_b.r = zeros(3);
  frame_b.R = identity(3);
end InertialSystem;
]
```


Chapter 10

Arrays

An array can be regarded as a collection of values, all of the same type. Modelica arrays can be multidimensional and are “rectangular,” which in the case of matrices has the consequence that all rows in a matrix have equal length, and all columns have equal length.

Each array has a certain dimensionality, i.e., number of dimensions. The degenerate case of a scalar variable is not really an array, but can be regarded as an array with zero dimensions. Vectors have one dimension, matrices have two dimensions, etc. *[So-called row vectors and column vectors do not exist in Modelica and cannot be distinguished since vectors have only one dimension. If distinguishing these is desired, row matrices and column matrices are available, being the corresponding two-dimensional entities. However, in practice this is seldom needed since the usual matrix arithmetic and linear algebra operations have been defined to give the expected behavior when operating on Modelica vectors and matrices.]*

Modelica is a strongly typed language, which also applies to array types. The number of dimensions of an array is fixed and cannot be changed at run-time *[in order to permit strong type checking and efficient implementation.]* However, the sizes of array dimensions can be computed at run-time, *[allowing fairly generic array manipulation code to be written as well as interfacing to standard numeric libraries implemented in other programming languages.]*

An array is allocated by declaring an array variable or calling an array constructor. Elements of an array can be indexed by Integer, Boolean, or enumeration values.

10.1 Array Declarations

The Modelica type system includes scalar number, vector, matrix (number of dimensions, $\text{ndim}=2$), and arrays of more than two dimensions. *[There is no distinguishing between a row and column vector.]*

The following table shows the two possible forms of declarations and defines the terminology. C is a placeholder for any class, including the built-in type classes Real, Integer, Boolean, String, and enumeration types. The type of a dimension upper bound expression, e.g. n, m, p,... in the table below, need to be a subtype of Integer or the name E for an enumeration type E, or Boolean. Colon (:) indicates that the dimension upper bound is unknown and is a subtype of Integer.

Upper and lower array dimension index bounds are described in Section 10.1.1.

An array indexed by Boolean or enumeration type can only be used in the following ways:

- Subscripted using expressions of the appropriate type (i.e. Boolean or the enumerated type)
- Binding equations of the form $x_1 = x_2$ as well as declaration assignments of the form $x_1 := x_2$ are allowed for arrays independent of whether the index types of dimensions are subtypes of Integer, Boolean, or enumeration types.

Table 10-1. General forms of declaration of arrays.

<i>Modelica form 1</i>	<i>Modelica form 2</i>	<i># dimensions</i>	<i>Designation</i>	<i>Explanation</i>
C x;	C x;	0	Scalar	Scalar
C[n] x;	C x[n];	1	Vector	n – Vector

$C[E] \ x;$	$C \ x[E]$	1	Vector	Vector index by enumeration type E
$C[n, m] \ x;$	$C \ x[n, m];$	2	Matrix	$n \times m$ Matrix
$C[n_1, n_2, \dots, n_k] \ x;$	$C \ x[n_1, n_2, \dots, n_k];$	k	Array	Array with k dimensions ($k \geq 0$).

[The number of dimensions and the dimensions sizes are part of the type, and shall be checked for example at redeclarations. Declaration form 1 displays clearly the type of an array, whereas declaration form 2 is the traditional way of array declarations in languages such as Fortran, C, C++.

```
Real[:] v1, v2          // vectors v1 and v2 have unknown sizes. The actual sizes may be different.
```

It is possible to mix the two declaration forms although it might be confusing.

```
Real[3,2] x[4,5];      // x has type Real[4,5,3,2];
```

The reason for this order is given by examples such as:

```
type R3=Real[3];
R3 a;
R3 b[1]={a};
Real[3] c[1]=b;
```

Using a type for “a” and “b” in this way is normal, and substituting a type by its definition allow “c”.

A vector y indexed by enumeration values

```
type TwoEnums = enumeration(one,two);
Real[TwoEnums] y;
]
```

Zero-valued dimensions are allowed, so: $C \times [0]$; declares an empty vector and: $C \times [0, 3]$; an empty matrix.

[Special cases:

Table 10-2. Declaration of arrays as 1-vectors, row-vectors, or column-vectors of arrays.

Modelica form 1	Modelica form 2	# dimensions	Designation	Explanation
$C[1] \ x;$	$C \ x[1];$	1	Vector	1 – Vector, representing a scalar
$C[1,1] \ x;$	$C \ x[1, 1];$	2	Matrix	1 x 1 – Matrix, representing a scalar
$C[n,1] \ x;$	$C \ x[n, 1];$	2	Matrix	$n \times 1$ – Matrix, representing a column
$C[1,n] \ x;$	$C \ x[1, n];$	2	Matrix	1 x n – Matrix, representing a row

]

The type of an array of array is the multidimensional array which is constructed by taking the first dimensions from the component declaration and subsequent dimensions from the maximally expanded component type. A type is maximally expanded, if it is either one of the built-in types (Real, Integer, Boolean, String, enumeration type) or it is not a type class. Before operator overloading is applied, a type class of a variable is maximally expanded.

[Example:

```
type Voltage = Real(unit = "V");
type Current = Real(unit = "A");
connector Pin
  Voltage v;          // type class of v = Voltage, type of v = Real
  flow Current i;      // type class of i = Current, type of i = Real
end Pin;
type MultiPin = Pin[5];

MultiPin[4] p;         // type class of p is MultiPin, type of p is Pin[4,5];

type Point = Real[3];
Point p1[10];
Real p2[10,3];
```

The components p1 and p2 have identical types.

```

p2[5] = p1[2] + p2[4];    // equivalent to    p2[5,:] = p1[2,:] + p2[4,:]
Real r[3] = p1[2];        // equivalent to    r[3] = p1[2,:]
]

```

[Automatic assertions at simulation time:

Let A be a declared array and i be the declared maximum dimension size of the d_i -dimension, then an assert statement `assert(i >= 0, ...)` is generated provided this assertion cannot be checked at compile time. It is a quality of implementation issue to generate a good error message if the assertion fails.

Let A be a declared array and i be an index accessing an index of the d_i -dimension. Then for every such index-access an assert statement `assert(i >= 1 and i <= size(A, di), ...)` is generated, provided this assertion cannot be checked at compile time.

For efficiency reasons, these implicit assert statement may be optionally suppressed.]

10.1.1 Array Dimension Lower and Upper Index Bounds

The lower and upper index bounds for a dimension of an array indexed by Integer, Boolean, or enumeration values are as follows:

- An array dimension indexed by integers has a lower bound of 1 and an upper bound being the size of the dimension.
- An array dimension indexed by Boolean values has the lower bound `false` and the upper bound `true`.
- An array dimension indexed by enumeration values of the type `E=enumeration(e1, e2, ..., en)` has the lower bound `E.e1` and the upper bound `E.en`.

10.2 Flexible Array Sizes

Regarding flexible array sizes and resizing of arrays in functions, see Section 12.4.4.

10.3 Built-in Array Functions

Modelica provides a number of built-in functions that are applicable to arrays.

The following `promote` function cannot be used in Modelica, but is utilized below to define other array operators and functions:

<code>promote(A, n)</code>	Fills dimensions of size 1 from the right to array A upto dimension n , where " $n \geq \text{ndims}(A)$ " is required. Let $C = \text{promote}(A, n)$, with $nA = \text{ndims}(A)$, then $\text{ndims}(C) = n$, $\text{size}(C, j) = \text{size}(A, j)$ for $1 \leq j \leq nA$, $\text{size}(C, j) = 1$ for $nA + 1 \leq j \leq n$, $C[i_1, \dots, i_{nA}, 1, \dots, 1] = A[i_1, \dots, i_{nA}]$
----------------------------	--

[The function `promote` cannot be used in Modelica, because the number of dimensions of the returned array cannot be determined at compile time if n is a variable. Below, `promote` is only used for constant n .

Some examples of using the functions defined in the following Section 10.3.1 to Section 10.3.5:

```

Real x[4,1,6];
size(x,1) = 4;
size(x);          // vector with elements 4, 1, 6
size(2*x+x) = size(x);
Real[3] v1 = fill(1.0, 3);
Real[3,1] m = matrix(v1);
Real[3] v2 = vector(m);
Boolean check[3,4] = fill(true, 3, 4);
]

```

10.3.1 Array Dimension and Size Functions

The following built-in functions for array dimensions and dimension sizes are provided:

Table 10-3. Built-in array dimension and size functions.

<i>Modelica</i>	<i>Explanation</i>
<code>ndims(A)</code>	Returns the number of dimensions k of array expression A , with $k \geq 0$.
<code>size(A, i)</code>	Returns the size of dimension i of array expression A where i shall be > 0 and $\leq \text{ndims}(A)$.
<code>size(A)</code>	Returns a vector of length <code>ndims(A)</code> containing the dimension sizes of A .

10.3.2 Dimensionality Conversion Functions

The following built-in conversion functions convert scalars, vectors, and arrays to scalars, vectors, or matrices by adding or removing 1-sized dimensions.

Table 10-4. Built-in dimensionality conversion functions.

<i>Modelica</i>	<i>Explanation</i>
<code>scalar(A)</code>	Returns the single element of array A . <code>size(A, i) = 1</code> is required for $1 \leq i \leq \text{ndims}(A)$.
<code>vector(A)</code>	Returns a 1-vector, if A is a scalar and otherwise returns a vector containing all the elements of the array, provided there is at most one dimension size > 1 .
<code>matrix(A)</code>	Returns <code>promote(A, 2)</code> , if A is a scalar or vector and otherwise returns the elements of the first two dimensions as a matrix. <code>size(A, i) = 1</code> is required for $2 < i \leq \text{ndims}(A)$.

10.3.3 Specialized Array Constructor Functions

An array constructor function constructs and returns an array computed from its arguments. Most of the constructor functions in the table below construct an array by filling in values according to a certain pattern, in several cases just giving all array elements the same value. The general array constructor with syntax `array(...)` or `{...}` is described in Section 10.4.

Table 10-5. Specialized array constructor functions.

<i>Modelica</i>	<i>Explanation</i>
<code>identity(n)</code>	Returns the $n \times n$ Integer identity matrix, with ones on the diagonal and zeros at the other places.
<code>diagonal(v)</code>	Returns a square matrix with the elements of vector v on the diagonal and all other elements zero.
<code>zeros(n₁, n₂, n₃, ...)</code>	Returns the $n_1 \times n_2 \times n_3 \times \dots$ Integer array with all elements equal to zero ($n_i \geq 0$).
<code>ones(n₁, n₂, n₃, ...)</code>	Return the $n_1 \times n_2 \times n_3 \times \dots$ Integer array with all elements equal to one ($n_i \geq 0$).
<code>fill(s, n₁, n₂, n₃, ...)</code>	Returns the $n_1 \times n_2 \times n_3 \times \dots$ array with all elements equal to scalar or array expression s ($n_i \geq 0$). The returned array has the same type as s . Recursive definition: <code>fill(s, n₁, n₂, n₃, ...) = fill(fill(s, n₂, n₃, ...), n₁)</code> ; <code>fill(s, n) = {s, s, ..., s}</code>
<code>linspace(x1, x2, n)</code>	Returns a Real vector with n equally spaced elements, such that <code>v=linspace(x1, x2, n)</code> , <code>v[i] = x1 + (x2-x1)*(i-1)/(n-1)</code> for $1 \leq i \leq n$. It is required that $n \geq 2$. The

	arguments x1 and x2 shall be numeric scalar expressions.
--	--

10.3.4 Reduction Functions and Operators

A reduction function “reduces” an array (or several scalars) to one value (normally a scalar). The following reduction functions are available:

Table 10-6. Array reduction functions and operators.

<i>Modelica</i>	<i>Explanation</i>
<code>min(A)</code>	Returns the smallest element of array expression A.
<code>min(x,y)</code>	Returns the smallest element of the scalars x and y.
<code>min(e(i, ..., j) for i in u, ..., j in v)</code>	Also described in Section 10.3.4.1 Returns the smallest value of the scalar expression e(i, ..., j) evaluated for all combinations of i in u, ..., j in v:
<code>max(A)</code>	Returns the largest element of array expression A.
<code>max(x,y)</code>	Returns the largest element of the scalars x and y.
<code>max(e(i, ..., j) for i in u, ..., j in v)</code>	Also described in Section 10.3.4.1 Returns the largest value of the scalar expression e(i, ..., j) evaluated for all combinations of i in u, ..., j in v:
<code>sum(A)</code>	Returns the scalar sum of all the elements of array expression: A[1,...,1]+A[2,...,1]+...+A[end,...,1]+A[end,...,end]
<code>sum(e(i, ..., j) for i in u, ..., j in v)</code>	Also described in Section 10.3.4.1 Returns the sum of the expression e(i, ..., j) evaluated for all combinations of i in u, ..., j in v: e(u[1],...,v[1])+e(u[2],...,v[1])+...+e(u[end],...,v[1])+...+e(u[end],...,v[end]) The type of <code>sum(e(i, ..., j) for i in u, ..., j in v)</code> is the same as the type of e(i,...j).
<code>product(A)</code>	Returns the scalar product of all the elements of array expression A. A[1,...,1]*A[2,...,1]*...*A[end,...,1]*A[end,...,end]
<code>product(e(i, ..., j) for i in u, ..., j in v)</code>	Also described in Section 10.3.4.1. Returns the product of the scalar expression e(i, ..., j) evaluated for all combinations of i in u, ..., j in v: e(u[1],...,v[1])*e(u[2],...,v[1])*...*(u[end],...,v[1])*...*(u[end],...,v[end]) The type of <code>product(e(i, ..., j) for i in u, ..., j in v)</code> is the same as the type of e(i,...j).

10.3.4.1 Reduction Expressions

An expression:

```
function-name "(" expression1 for iterators ")"
```

is a reduction-expression. The expressions in the iterators of a reduction-expression shall be vector expressions. They are evaluated once for each reduction-expression, and are evaluated in the scope immediately enclosing the reduction-expression.

For an iterator:

```
IDENT in expression2
```

the loop-variable, IDENT, is in scope inside expression1. The loop-variable may hide other variables, as in for-clauses. The result depends on the function-name, and currently the only legal function-names are the built-in operators array, sum, product, min, and max. For array, see Section 10.4. If function-name is sum,

product, min, or max the result is of the same type as `expression1` and is constructed by evaluating `expression1` for each value of the loop-variable and computing the sum, product, min, or max of the computed elements. For deduction of ranges, see Section 11.2.2.1.

Function-name	Restriction on expression1	Result if expression2 is empty
sum	None	zeros(...)
product	Scalar	1
min	Scalar	Modelica.Constants.inf
max	Scalar	-Modelica.Constants.inf

[Example:

```

sum(i for i in 1:10)           // Gives  $\sum_{i=1}^{10} i = 1+2+\dots+10=55$ 
// Read it as: compute the sum of i for i in the range 1 to 10.

sum(i^2 for i in {1,3,7,6})    // Gives  $\sum_{i \in \{1, 3, 7, 6\}} i^2 = 1+9+49+36=95$ 

{product(j for j in 1:i) for i in 0:4} // Gives {1,1,2,6,24}
max(i^2 for i in {3,7,6})      // Gives 49
]
```

10.3.5 Matrix and Vector Algebra Functions

The following set of built-in matrix and vector algebra functions are available:

Table 10-7. Matrix and vector algebra functions.

Modelica	Explanation
<code>transpose(A)</code>	Permutes the first two dimensions of array A. It is an error, if array A does not have at least 2 dimensions.
<code>outerProduct(v1,v2)</code>	Returns the outer product of vectors v1 and v2 (= <code>matrix(v)*transpose(matrix(v))</code>).
<code>symmetric(A)</code>	Returns a matrix where the diagonal elements and the elements above the diagonal are identical to the corresponding elements of matrix A and where the elements below the diagonal are set equal to the elements above the diagonal of A, i.e., $B := \text{symmetric}(A) \rightarrow B[i,j] := A[i,j]$, if $i \leq j$, $B[i,j] := A[j,i]$, if $i > j$.
<code>cross(x,y)</code>	Returns the cross product of the 3-vectors x and y, i.e. $\text{cross}(x,y) = \text{vector}([x[2]*y[3]-x[3]*y[2]; x[3]*y[1]-x[1]*y[3]; x[1]*y[2]-x[2]*y[1]])$;
<code>skew(x)</code>	Returns the 3 x 3 skew symmetric matrix associated with a 3-vector, i.e., $\text{cross}(x,y) = \text{skew}(x)*y$; $\text{skew}(x) = [0, -x[3], x[2]; x[3], 0, -x[1]; -x[2], x[1], 0]$;

10.4 Vector, Matrix and Array Constructors

The constructor function `array(A,B,C,...)` constructs an array from its arguments according to the following rules:

- Size matching: All arguments must have the same sizes, i.e., $\text{size}(A) = \text{size}(B) = \text{size}(C) = \dots$
- All arguments must be type compatible expressions (Section 6.6) giving the type of the elements. The data type of the result array is the maximally expanded type of the arguments. Real and Integer subtypes can be mixed resulting in a Real result array where the Integer numbers have been transformed to Real numbers.

- Each application of this constructor function adds a one-sized dimension to the left in the result compared to the dimensions of the argument arrays, i.e., $\text{ndims}(\text{array}(A,B,C)) = \text{ndims}(A) + 1 = \text{ndims}(B) + 1, \dots$
- $\{A, B, C, \dots\}$ is a shorthand notation for $\text{array}(A, B, C, \dots)$.
- There must be at least one argument [*i.e.*, $\text{array}()$ or $\{\}$ is not defined].

[Examples:

$\{1, 2, 3\}$ is a 3-vector of type Integer.

$\{\{11, 12, 13\}, \{21, 22, 23\}\}$ is a 2×3 matrix of type Integer

$\{\{1.0, 2.0, 3.0\}\}$ is a $1 \times 1 \times 3$ array of type Real.

```
Real[3] v = array(1, 2, 3.0);
type Angle = Real(unit="rad");
parameter Angle alpha = 2.0;      // type of alpha is Real.
// array(alpha, 2, 3.0) or {alpha, 2, 3.0} is a 3-vector of type Real.
Angle[3] a = {1.0, alpha, 4};     // type of a is Real[3].
]
```

10.4.1 Array Constructor with Iterators

An expression:

```
"{" expression for iterators "}"
```

or

```
array "(" expression for iterators ")"
```

is an array constructor with iterators. The expressions inside the iterators of an array constructor shall be vector expressions. They are evaluated once for each array constructor, and is evaluated in the scope immediately enclosing the array constructor.

For an iterator:

```
IDENT in array_expression
```

the loop-variable, IDENT, is in scope inside expression in the array construction. The loop-variable may hide other variables, as in for-clauses. The loop-variable has the same type as the type of the elements of array_expression. For deduction of ranges, see Section 11.2.2.1.

10.4.1.1 Array Constructor with One Iterator

If only one iterator is used, the result is a vector constructed by evaluating expression for each value of the loop-variable and forming an array of the result.

[Example:

```
array(i for i in 1:10)
// Gives the vector 1:10={1,2,3,...,10}

{r for r in 1.0 : 1.5 : 5.5}
// Gives the vector 1.0:1.5:5.5={1.0, 2.5, 4.0, 5.5}

{i^2 for i in {1,3,7,6}}
// Gives the vector {1, 9, 49, 36}
```

10.4.1.2 Array Constructor with Several Iterators

The notation with several iterators is a shorthand notation for nested array constructors. The notation can be expanded into the usual form by replacing each ',' by '}' for' and prepending the array constructor with a '{'.

[Example:

```
Real hilb[:, :] = {(1/(i+j-1) for i in 1:n, j in 1:n);
```

```
Real hilb2[:,:] = {{ (1/(i+j-1)) for j in 1:n } for i in 1:n};
```

10.4.2 Array Concatenation

The function `cat(k,A,B,C,...)` concatenates arrays A,B,C,... along dimension k according to the following rules:

- Arrays A, B, C, ... must have the same number of dimensions, i.e., $\text{ndims}(A) = \text{ndims}(B) = \dots$
- Arrays A, B, C, ... must be type compatible expressions (Section 6.6) giving the type of the elements of the result. The maximally expanded types should be equivalent. Real and Integer subtypes can be mixed resulting in a Real result array where the Integer numbers have been transformed to Real numbers.
- k has to characterize an existing dimension, i.e., $1 \leq k \leq \text{ndims}(A) = \text{ndims}(B) = \text{ndims}(C)$; k shall be an integer number.
- Size matching: Arrays A, B, C, ... must have identical array sizes with the exception of the size of dimension k, i.e., $\text{size}(A_j) = \text{size}(B_j)$, for $1 \leq j \leq \text{ndims}(A)$ and $j \neq k$.

[Examples:

```
Real [2,3] r1 = cat(1, {{1.0, 2.0, 3}}, {{4, 5, 6}});
Real [2,6] r2 = cat(2, r1, 2*r1);
]
```

Concatenation is formally defined according to:

Let $R = \text{cat}(k,A,B,C,\dots)$, and let $n = \text{ndims}(A) = \text{ndims}(B) = \text{ndims}(C) = \dots$, then

$\text{size}(R,k) = \text{size}(A,k) + \text{size}(B,k) + \text{size}(C,k) + \dots$

$\text{size}(R,j) = \text{size}(A,j) = \text{size}(B,j) = \text{size}(C,j) = \dots$, for $1 \leq j \leq n$ and $j \neq k$.

$R[i_1, \dots, i_k, \dots, i_n] = A[i_1, \dots, i_k, \dots, i_n]$, for $i_k \leq \text{size}(A,k)$,

$R[i_1, \dots, i_k, \dots, i_n] = B[i_1, \dots, i_k - \text{size}(A,k), \dots, i_n]$, for $i_k \leq \text{size}(A,k) + \text{size}(B,k)$,

....

where $1 \leq i_j \leq \text{size}(R,j)$ for $1 \leq j \leq n$.

10.4.2.1 Array Concatenation along First and Second Dimensions

For convenience, a special syntax is supported for the concatenation along the first and second dimensions.

- *Concatenation along first dimension:*
 $[A; B; C; \dots] = \text{cat}(1, \text{promote}(A,n), \text{promote}(B,n), \text{promote}(C,n), \dots)$ where
 $n = \max(2, \text{ndims}(A), \text{ndims}(B), \text{ndims}(C), \dots)$. If necessary, 1-sized dimensions are added to the right of A, B, C before the operation is carried out, in order that the operands have the same number of dimensions which will be at least two.
- *Concatenation along second dimension:*
 $[A, B, C, \dots] = \text{cat}(2, \text{promote}(A,n), \text{promote}(B,n), \text{promote}(C,n), \dots)$ where
 $n = \max(2, \text{ndims}(A), \text{ndims}(B), \text{ndims}(C), \dots)$. If necessary, 1-sized dimensions are added to the right of A, B, C before the operation is carried out, especially that each operand has at least two dimensions.
- The two forms can be mixed. $[A, B; C, D]$ has higher precedence than $[A; B, C; D]$, e.g., $[a, b; c, d]$ is parsed as $[[a,b]; [c,d]]$.
- $[A] = \text{promote}(A, \max(2, \text{ndims}(A)))$, i.e., $[A] = A$, if A has 2 or more dimensions, and it is a matrix with the elements of A, if A is a scalar or a vector.
- There must be at least one argument (i.e. $[]$ is not defined)

[Examples:

```
Real s1, s2, v1[n1], v2[n2], M1[m1,n],
      M2[m2,n], M3[n,m1], M4[n,m2], K1[m1,n,k], K2[m2,n,k];
```

$[v1; v2]$ is a $(n1+n2) \times 1$ matrix

$[M1; M2]$ is a $(m1+m2) \times n$ matrix

```

[M3,M4] is a n x (m1+m2) matrix
[K1;K2] is a (m1+m2) x n x k array
[s1;s2] is a 2 x 1 matrix
[s1,s1] is a 1 x 2 matrix
[s1] is a 1 x 1 matrix
[v1] is a n1 x 1 matrix

Real[3] v1 = array(1, 2, 3);
Real[3] v2 = {4, 5, 6};
Real[3,2] m1 = [v1, v2];
Real[3,2] m2 = [v1, [4;5;6]]; // m1 = m2
Real[2,3] m3 = [1, 2, 3; 4, 5, 6];
Real[1,3] m4 = [1, 2, 3];
Real[3,1] m5 = [1; 2; 3];
]

```

10.4.3 Vector Construction

Vectors can be constructed with the general array constructor, e.g., `Real[3] v = {1,2,3}`.

The range vector operator or colon operator of simple-expression can be used instead of or in combination with this general constructor to construct Real, Integer, Boolean or enumeration type vectors. Semantics of the colon operator:

- `j : k` is the Integer vector $\{j, j+1, \dots, k\}$, if `j` and `k` are of type Integer.
- `j : k` is the Real vector $\{j, j+1.0, \dots, n\}$, with $n = \text{floor}(k-j)$, if `j` and/or `k` are of type Real.
- `j : k` is a Real, Integer, Boolean, or enumeration type vector with zero elements, if `j > k`.
- `j : d : k` is the Integer vector $\{j, j+d, \dots, j+n*d\}$, with $n = (k-j)/d$, if `j`, `d`, and `k` are of type Integer.
- `j : d : k` is the Real vector $\{j, j+d, \dots, j+n*d\}$, with $n = \text{floor}((k-j)/d)$, if `j`, `d`, or `k` are of type Real.
- `j : d : k` is a Real or Integer vector with zero elements, if `d > 0` and `j > k` or if `d < 0` and `j < k`.
- `false : true` is the Boolean vector $\{\text{false}, \text{true}\}$.
- `j:j` is $\{j\}$ if `j` is Real, Integer, Boolean, or enumeration type.
- `E.ei : E.ej` is the enumeration type vector $\{E.ei, \dots, E.ej\}$ where `E.ej > E.ei`, and `ei` and `ej` belong to some enumeration type `E=enumeration(...ei,...ej,...)`.

[Examples:

```

Real v1[5] = 2.7 : 6.8;
Real v2[5] = {2.7, 3.7, 4.7, 5.7, 6.7}; // = same as v1
Boolean b1[2] = false:true;
Colors = enumeration(red,blue,green);
Colors ec[3] = Colors.red : Colors.green;
]

```

10.5 Array Indexing

The array indexing operator `name [...]` is used to access array elements for retrieval of their values or for updating these values. An indexing operation is subject to upper and lower array dimension index bounds (Section 10.1.1). [An indexing operation is assumed to take constant time, i.e., largely independent of the size of the array.] The indexing operator takes two or more operands, where the first operand is the array to be indexed and the rest of the operands are index expressions:

```
arrayname[indexexpr1, indexexpr2, ...]
```

A colon is used to denote all indices of one dimension. A vector expression can be used to pick out selected rows, columns and elements of vectors, matrices, and arrays. The number of dimensions of the expression is reduced by the number of scalar index arguments.

It is also possible to use the array access operator to assign to element/elements of an array in algorithm sections. If the index is an array the assignments take place in the order given by the index array. For assignments to arrays and elements of arrays, the entire right-hand side and the index on the left-hand side is evaluated before any element is assigned a new value.

[Examples:

$a[:, j]$ is a vector of the j -th column of a ,

$a[j : k]$ is $\{a[j], a[j+1], \dots, a[k]\}$

$a[:, j : k]$ is $[a[:, j], a[:, j+1], \dots, a[:, k]]$,

$v[2:2:8] = v[\{2, 4, 6, 8\}]$.

$v[\{j, k\}] := \{2, 3\}$; // Same as $v[j] := 2$; $v[k] := 3$;

$v[\{1, 1\}] := \{2, 3\}$; // Same as $v[1] := 3$;

if x is a vector, $x[1]$ is a scalar, but the slice $x[1:5]$ is a vector (a vector-valued or colon index expression causes a vector to be returned).

]

[Examples given the declaration $x[n, m]$, $v[k]$, $z[i, j, p]$:

Table 10-8. Examples of scalars vs. array slices created with the colon index.

Expression	# dimensions	Type of value
$x[1, 1]$	0	Scalar
$x[:, 1]$	1	n – Vector
$x[1, :]$	1	m – Vector
$v[1:p]$	1	p – Vector
$x[1:p, :]$	2	$p \times m$ – Matrix
$x[1:1, :]$	2	$1 \times m$ - "row" matrix
$x[\{1, 3, 5\}, :]$	2	$3 \times m$ – Matrix
$x[:, v]$	2	$n \times k$ – Matrix
$z[:, 3, :]$	2	$i \times p$ – Matrix
$x[\text{scalar}([1]), :]$	1	m – Vector
$x[\text{vector}([1]), :]$	2	$1 \times m$ - "row" matrix

]

10.5.1 Indexing with Boolean or Enumeration Values

Arrays can be indexed using values of enumeration types or the Boolean type, not only by integers.

[Example:

```

type ShirtSizes = enumeration(small, medium, large, xlarge);
Real[ShirtSizes] w;
Real[Boolean] b2;
algorithm
  w[ShirtSizes.large] := 2.28; // Assign a value to an element of w
  b2[true] := 10.0;

```

]

10.5.2 Indexing with end

The expression `end` may only appear inside array subscripts, and if used in the i :th subscript of an array expression `A` it is equivalent to `size(A,i)` provided indices to `A` are a subtype of Integer. If used inside nested array subscripts it refers to the most closely nested array.

[Examples:

```
A[end-1,end] is A[size(A,1)-1,size(A,2)]

A[v[end],end] is A[v[size(v,1)],size(A,2)] // since the first end is referring to end of v.
]
```

10.6 Scalar, Vector, Matrix, and Array Operator Functions

The mathematical operations defined on scalars, vectors, and matrices are the subject of linear algebra.

In all contexts that require an expression which is a subtype of Real, an expression which is a subtype of Integer can also be used; the Integer expression is automatically converted to Real.

The term numeric or numeric class is used below for a subtype of the Real or Integer type classes.

10.6.1 Equality and Assignment

Equality `a=b` and assignment `a:=b` of scalars, vectors, matrices, and arrays is defined element-wise and require both objects to have the same number of dimensions and corresponding dimension sizes. The operands need to be type equivalent. This is legal for the simple types and all types satisfying the requirements for a record, and is in the latter case applied to each component-element of the records.

Table 10-9. Equality and assignment of arrays and scalars.

Type of a	Type of b	Result of $a = b$	Operation ($j=1:n, k=1:m$)
Scalar	Scalar	Scalar	$a = b$
Vector[n]	Vector[n]	Vector[n]	$a[j] = b[j]$
Matrix[n, m]	Matrix[n, m]	Matrix[n, m]	$a[j, k] = b[j, k]$
Array[n, m, ...]	Array[n, m, ...]	Array[n, m, ...]	$a[j, k, ...] = b[j, k, ...]$

10.6.2 Array Element-wise Addition, Subtraction, and String Concatenation

Addition `a+b` and subtraction `a-b` of numeric scalars, vectors, matrices, and arrays is defined element-wise and require `size(a)=size(b)` and a numeric type for `a` and `b`. Addition `a+b` of string scalars, vectors, matrices, and arrays is defined as element-wise string concatenation of corresponding elements from `a` and `b`, and require `size(a)=size(b)`.

Table 10-10. Array addition, subtraction, and string concatenation.

Type of a	Type of b	Result of $a +/- b$	Operation $c := a +/- b$ ($j=1:n, k=1:m$)
Scalar	Scalar	Scalar	$c := a +/- b$
Vector[n]	Vector[n]	Vector[n]	$c[j] := a[j] +/- b[j]$
Matrix[n, m]	Matrix[n, m]	Matrix[n, m]	$c[j, k] := a[j, k] +/- b[j, k]$
Array[n, m, ...]	Array[n, m, ...]	Array[n, m, ...]	$c[j, k, ...] := a[j, k, ...] +/- b[j, k, ...]$

Element-wise addition `a . + b` and subtraction `a . - b` of numeric scalars, vectors, matrices or arrays `a` and `b` requires a numeric type class for `a` and `b` and either `size(a) = size(b)` or scalar `a` or scalar `b`. Element-wise addition `a . + b` of string scalars, vectors, matrices, and arrays is defined as element-wise string concatenation of corresponding elements from `a` and `b`, and require either `size(a) = size(b)` or scalar `a` or scalar `b`.

Table 10-11 Array element-wise addition, subtraction, and string concatenation.

<i>Type of a</i>	<i>Type of b</i>	<i>Result of a .+/. - b</i>	<i>Operation c := a .+/. b (j=1:n, k=1:m)</i>
Scalar	Scalar	Scalar	c := a +/- b
Scalar	Array[n, m, ...]	Array[n, m, ...]	c[j, k, ...] := a +/- b[j, k, ...]
Array[n, m, ...]	Scalar	Array[n, m, ...]	c[j, k, ...] := a[j, k, ...] +/- b
Array[n, m, ...]	Array[n, m, ...]	Array[n, m, ...]	c [j, k, ...] := a[j, k, ...] +/- b[j, k, ...]

10.6.3 Array Element-wise Multiplication

Scalar multiplication $s * a$ or $a * s$ with numeric scalar s and numeric scalar, vector, matrix or array a is defined element-wise:

Table 10-12. Scalar and scalar to array multiplication of numeric elements

Type of s	Type of a	Type of $s^* a$ and $a^* s$	Operation $c := s^* a$ or $c := a^* s$ ($j=1:n, k=1:m$)
Scalar	Scalar	Scalar	$c := s^* a$
Scalar	Vector [n]	Vector [n]	$c[j] := s^* a[j]$
Scalar	Matrix [n, m]	Matrix [n, m]	$c[j, k] := s^* a[j, k]$
Scalar	Array[n, m, ...]	Array [n, m, ...]	$c[j, k, ...] := s^* a[j, k, ...]$

Element-wise multiplication `a.*b` of numeric scalars, vectors, matrices or arrays `a` and `b` requires a numeric type class for `a` and `b` and either `size(a) = size(b)` or scalar `a` or scalar `b`.

Table 10-13 Array element-wise multiplication

<i>Type of a</i>	<i>Type of b</i>	<i>Type of a .* b</i>	<i>Operation c:=a .* b (j=1:n, k=1:m)</i>
Scalar	Scalar	Scalar	$c := a * b$
Scalar	Array[n, m, ...]	Array[n, m, ...]	$c[j, k, \dots] := a * b[j, k, \dots]$
Array[n, m, ...]	Scalar	Array[n, m, ...]	$c[j, k, \dots] := a[j, k, \dots] * b$
Array[n, m, ...]	Array[n, m, ...]	Array [n, m, ...]	$c[j, k, \dots] := a[j, k, \dots] * b[j, k, \dots]$

10.6.4 Matrix and Vector Multiplication of Numeric Arrays

Multiplication $a*b$ of numeric vectors and matrices is defined only for the following combinations:

Table 10-14. Matrix and vector multiplication of arrays with numeric elements.

<i>Type of a</i>	<i>Type of b</i>	<i>Type of a*b</i>	<i>Operation c := a*b</i>
Vector [n]	Vector [n]	Scalar	$c := \text{sum}_k(a[k]*b[k]), k=1:n$
Vector [n]	Matrix [n, m]	Vector [m]	$c[j] := \text{sum}_k(a[k]*b[k, j]), j=1:m, k=1:n$
Matrix [n, m]	Vector [m]	Vector [n]	$c[j] := \text{sum}_k(a[j, k]*b[k])$
Matrix [n, m]	Matrix [m, p]	Matrix [n, p]	$c[i, j] = \text{sum}_k(a[i, k]*b[k, j]), i=1:n, k=1:m, j=1:p$

[*Example:*

[illegible]

```

[v]*transpose([v])      // outer product
v*A*v                  // scalar
transpose([v])*A*v      // vector with one element
]

```

10.6.5 Division of Scalars or Numeric Arrays by Numeric Scalars

Division a/s of numeric scalars, vectors, matrices, or arrays a and numeric scalars s is defined element-wise. The result is always of real type. In order to get integer division with truncation use the function `div`.

Table 10-15. Division of scalars and arrays by numeric elements.

Type of a	Type of s	Result of a/s	Operation $c := a/s$ ($j=1:n, k=1:m$)
Scalar	Scalar	Scalar	$c := a/s$
Vector[n]	Scalar	Vector[n]	$c[k] := a[k]/s$
Matrix[n, m]	Scalar	Matrix[n, m]	$c[j, k] := a[j, k]/s$
Array[n, m, ...]	Scalar	Array[n, m, ...]	$c[j, k, ...] := a[j, k, ...]/s$

10.6.6 Array Element-wise Division

Element-wise division $a ./ b$ of numeric scalars, vectors, matrices or arrays a and b requires a numeric type class for a and b and either $\text{size}(a) = \text{size}(b)$ or scalar a or scalar b .

Table 10-16 Element-wise division of arrays

Type of a	Type of b	Type of $a ./ b$	Operation $c := a ./ b$ ($j=1:n, k=1:m$)
Scalar	Scalar	Scalar	$c := a/b$
Scalar	Array[n, m, ...]	Array[n, m, ...]	$c[j, k, ...] := a/b[j, k, ...]$
Array[n, m, ...]	Scalar	Array[n, m, ...]	$c[j, k, ...] := a[j, k, ...]/b$
Array[n, m, ...]	Array[n, m, ...]	Array[n, m, ...]	$c[j, k, ...] := a[j, k, ...]/b[j, k, ...]$

[Element-wise division by scalar ($./$) and division by scalar ($/$) are identical: $a./s = a/s$.

Example:

```

2./[1,2;3,4]    // error, since 2.0/[1,2;3,4]
2 ./[1,2;3,4]   // fine, element-wise division

```

This is a consequence of the parsing rules, since `2.` is a lexical unit. Using a space after the literal solves the problem.]

10.6.7 Exponentiation of Scalars of Numeric Elements

Exponentiation a^b is defined as `pow(double a, double b)` in the ANSI C library if both `"a"` and `"b"` are Real scalars. A Real scalar value is returned. If `"a"` or `"b"` are Integer scalars, they are automatically promoted to "Real". Consequences of exceptional situations, such as ($a=0.0$ and $b \leq 0.0$, $a < 0$ and b is not an integer) or overflow are undefined

Element-wise exponentiation $a.^b$ of numeric scalars, vectors, matrices, or arrays a and b requires a numeric type class for a and b and either $\text{size}(a) = \text{size}(b)$ or scalar a or scalar b .

Table 10-17 Element-wise exponentiation of arrays

Type of a	Type of b	Type of $a.^b$	Operation $c := a.^b$ ($j=1:n, k=1:m$)
-------------	-------------	----------------	--

Scalar	Scalar	Scalar	$c := a^b$
Scalar	Array[n, m, ...]	Array[n, m, ...]	$c[j, k, \dots] := a^b[j, k, \dots]$
Array[n, m, ...]	Scalar	Array[n, m, ...]	$c[j, k, \dots] := a[j, k, \dots]^b$
Array[n, m, ...]	Array[n, m, ...]	Array[n, m, ...]	$c[j, k, \dots] := a[j, k, \dots]^b[j, k, \dots]$

[Example:

```
2.^[1,2;3,4]    // error, since 2.0^[1,2;3,4]
2 .^[1,2;3,4]   // fine, element wise exponentiation
```

This is a consequence of the parsing rules, i.e. since 2. could be a lexical unit it seen as a lexical unit; using a space after literals solves the problem.]

10.6.8 Scalar Exponentiation of Square Matrices of Numeric Elements

Exponentiation a^s is defined if a is a square numeric matrix and s is a scalar as a subtype of Integer with $s \geq 0$. The exponentiation is done by repeated multiplication

(e.g.:

```
a^3 = a*a*a; a^0 = identity(size(a,1));
assert(size(a,1)==size(a,2),"Matrix must be square");
a^1 = a;
```

[Non-Integer exponents are forbidden, because this would require to compute the eigenvalues and eigenvectors of "a" and this is no longer an elementary operation].

10.6.9 Slice Operation

The following holds for slice operations:

- If a is an array containing scalar components and m is a component of those components, the expression $a.m$ is interpreted as a slice operation. It returns the array of components $\{a[1].m, \dots\}$.
- If m is also an array component, the slice operation is valid only if $\text{size}(a[1].m) = \text{size}(a[2].m) = \dots$
- The slicing operation can be combined with indexing, e.g. $a.m[1]$. It returns the array of components $\{a[1].m[1], a[2].m[1], \dots\}$, and does not require that $\text{size}(a[1].m) = \text{size}(a[2].m)$. The number of subscripts on m must exactly correspond to the number of array dimension for m , and is only valid if $\text{size}(a[1].m[\dots]) = \text{size}(a[2].m[\dots])$.

[Example: The size-restriction on the operand is only applicable if the indexing on the second operand uses vectors or colon as in the example:

```
constant Integer m=3;
Modelica.Blocks.Continuous.LowpassButterworth tf[m] (n=2:(m+1));
Real y[m];
Real y2,y3;
equation
  // Extract the x1 slice even though different x1's have different lengths
  y=tf.x1[1] ; // Legal, ={tf[1].x1[1], tf[2].x1[1], ... tf[m].x1[1]};

  y2=sum(tf.x1[:]); // Illegal to extract all elements since they have
                    // different lengths. Does not satisfy:
                    // size(tf[1].x1[:])=size(tf[2].x1[:])=... =size(tf[m].x1[:])

  y3=sum(tf.x1[1:2]); // Legal.
  // Since x1 has at least 2 elements in all tf, and
  // size(tf[1].x1[1:2])=size(tf[2].x1[1:2])=... =size(tf[m].x1[1:2])={2}
```


In this example the different $x1$ vectors have different lengths, but it is still possible to perform some operations on them.]

10.6.10 Relational Operators

Relational operators $<$, $<=$, $>$, $>=$, $==$, $<>$, are only defined for scalar operands of simple types, not for arrays, see Section 3.5

10.6.11 Boolean Operators

The operators, `and` and `or` take expressions of Boolean type, which are either scalars or arrays of matching dimensions. The operator `not` takes an expression of Boolean type, which is either scalar or an array. The result is the element-wise logical operation. For short-circuit evaluation of `and` and `or` see Section 3.3.

10.6.12 Vectorized Calls of Functions

See Section 12.4.5.

10.7 Empty Arrays

Arrays may have dimension sizes of 0. E.g.

```
Real x[0];           // an empty vector
Real A[0, 3], B[5, 0], C[0, 0]; // empty matrices
```

Empty matrices can be constructed with the `fill` function. E.g.

```
Real A[:, :] = fill(0.0, 0, 1); // a Real 0 x 1 matrix
Boolean B[:, :, :] = fill(false, 0, 1, 0); // a Boolean 0 x 1 x 0 matrix
```

It is not possible to access an element of an empty matrix, e.g. $v[j, k]$ cannot be evaluated if $v=[]$ because the assertion fails that the index must be bigger than one.

Size-requirements of operations, such as $+$, $-$, have also to be fulfilled if a dimension is zero. E.g.

```
Real[3,0] A, B;
Real[0,0] C;
A + B // fine, result is an empty matrix
A + C // error, sizes do not agree
```

Multiplication of two empty matrices results in a zero matrix of corresponding numeric type if the result matrix has no zero dimension sizes, i.e.,

```
Real[0,m]*Real[m,n] = Real[0,n] (empty matrix)
Real[m,n]*Real[n,0] = Real[m,0] (empty matrix)
Real[m,0]*Real[0,n] = fill(0.0, m, n) (non-empty matrix, with zero elements).
```

[Example:

```
Real u[p], x[n], y[q], A[n,n], B[n,p], C[q,n], D[q,p];
der(x) = A*x + B*u
      y = C*x + D*u
```

Assume $n=0, p>0, q>0$: Results in $y = D*u$

]

Chapter 11

Statements and Algorithm Sections

Whereas equations are very well suited for physical modeling, there are situations where computations are more conveniently expressed as algorithms, i.e., sequences of statements. In this chapter we describe the algorithmic constructs that are available in Modelica.

Statements are imperative constructs allowed in algorithm sections.

11.1 Algorithm Sections

Algorithm sections is comprised of the keyword `algorithm` followed by a sequence of statements. The formal syntax is as follows:

```
algorithm_section :
  [ initial ] algorithm { statement ";" | annotation ";" }
```

Equation equality = or any other kind of equation (see Chapter 8) shall not be used in an algorithm section.

11.1.1 Initial Algorithm Sections

See Section 8.6 for a description of both initial algorithm sections and initial equation sections.

11.2 Statements

Statements are imperative constructs allowed in algorithm sections. A flattened statement is identical to the corresponding nonflattened statement.

Names in statements are found as follows:

- If the name occurs inside an expression: it is first found among the lexically enclosing reduction functions (see Section 10.3.4) in order starting from the inner-most, and if not found it proceeds as if it were outside an expression:
- Names in a statement are first found among the lexically enclosing for-statements in order starting from the inner-most, and if not found:
- Names in a statement shall be found by looking up in the partially flattened enclosing class of the algorithm section.

The syntax of statements is as follows:

```
statement :
  ( component_reference ( "!=" expression | function_call_args )
  | "(" output_expression_list ")" "!=" component_reference function_call_args
  | break
  | return
  | if_statement
  | for_statement
  | while_statement
  | when_statement )
```

comment

11.2.1 Simple Assignment Statements

The syntax of simple assignment statement is as follows:

```
component_reference "[:=" expression
```

The `expression` is evaluated. The resulting value is stored into the variable denoted by `component_reference`.

11.2.1.1 Assignments from Called Functions with Multiple Results

There is a special form of assignment statement that is used only when the right-hand side contains a call to a function with multiple results. The left-hand side contains a parenthesized, comma-separated list of variables receiving the results from the function call. A function with n results needs $m \leq n$ receiving variables on the left-hand side.

```
(out1, out2, out3) := function_name(in1, in2, in3, in4);
```

It is possible to omit receiving variables from this list:

```
(out1,, out3) := function_name(in1, in2, in3, in4);
```

[*Example: The function `f` called below has three results and two inputs:*

```
(a, b, c) := f(1.0, 2.0);  
(x[1], x[2], x[3]) := f(3,4);  
]
```

The syntax of an assignment statement with a call to a function with multiple results is as follows:

```
(" output_expression_list ") "[:=" component_reference function_call_args
```

[*Also see Section 8.3.1 regarding calling functions with multiple results within equations.*]

11.2.2 For-statement

The syntax of a for-statement is as follows:

```
for for_indices loop  
  { statement ";" }  
end for
```

For-statements may optionally use several iterators (`for_indices`), see Section 11.2.2.2 for more information:

```
for_indices:  
  for_index {"," for_index}  
  
for_index:  
  IDENT [ in expression ]
```

The following is an example of a prefix of a for-statement:

```
for IDENT in expression loop
```

The expression of a for-statement shall be a vector expression. It is evaluated once for each for-statement, and is evaluated in the scope immediately enclosing the for-statement. The loop-variable (`IDENT`) is in scope inside the loop-construct and shall not be assigned to. The loop-variable has the same type as the type of the elements of the vector expression.

[*Example:*

```
for i in 1:10 loop           // i takes the values 1,2,3,...,10  
for r in 1.0 : 1.5 : 5.5 loop // r takes the values 1.0, 2.5, 4.0, 5.5
```

```

for i in {1,3,6,7} loop           // i takes the values 1, 3, 6, 7
for i in TwoEnums loop           // i takes the values TwoEnums.one, TwoEnums.two
                                     // for
TwoEnums = enumeration(one,two)

```

The loop-variable may hide other variables as in the following example. Using another name for the loop-variable is, however, strongly recommended.

```

constant Integer j=4;
Real x[j];
equation
for j in 1:j loop // The loop-variable j takes the values 1,2,3,4
  x[j]=j;          // Uses the loop-variable j
end for;
]

```

11.2.2.1 Implicit Iteration Ranges

An iterator IDENT in range-expr without the in range-expr requires that the IDENT appears as the subscript of one or several subscripted expressions. The dimension size of the array expression in the indexed position is used to deduce the range-expr as 1:size(array-expression,indexpos) if the indices are a subtype of Integer, or as E.e1:E.en if the indices are of an enumeration type E=enumeration(e1, ..., en), or as false:true if the indices are of type Boolean. If it is used to subscript several expressions, their ranges must be identical. The IDENT may also, inside a reduction-expression, array constructor expression, for-statement, or for-equation, occur freely outside of subscript positions, but only as a reference to the variable IDENT, and not for deducing ranges.

[Example:

```

Real x[4];
Real xsquared[:]={x[i]*x[i] for i};
// Same as: {x[i]*x[i] for i in 1:size(x,1)}
Real xsquared2[size(x,1)];
equation
for i loop // Same as: for i in 1:size(x,1) loop ...
  xsquared2[i]=x[i]^2;
end for;

type FourEnums=enumeration(one,two,three,four);
Real xe[FourEnums]= x;
Real xsquared3[FourEnums]={xe[i]*xe[i] for i};
Real xsquared4[FourEnums]={xe[i]*xe[i] for i in FourEnums};
Real xsquared5[FourEnums]={x[i]*x[i] for i};
]

```

The size of an array – the iteration range is evaluated on entry to the for-loop and the array size may not change during the execution of the for-loop.

11.2.2.2 Nested For-Loops and Reduction Expressions with Multiple Iterators

The notation with several iterators is a shorthand notation for nested for-statements or for-equations (or reduction-expressions). For for-statements or for-equations it can be expanded into the usual form by replacing each “,” by ‘loop for’ and adding extra ‘end for’. For reduction-expressions it can be expanded into the usual form by replacing each ‘,’ by ‘) for’ and prepending the reduction-expression with ‘function-name(‘.

[Example:

```

Real x[4,3];
algorithm
for j, i in 1:2 loop
  // The loop-variable j takes the values 1,2,3,4 (due to use)
  // The loop-variable i takes the values 1,2 (given range)
  x[j,i] := j+i;
end for;
]

```

11.2.3 While-Statement

The while-statement has the following syntax:

```
while expression loop
{ statement ";" }
end while
```

The expression of a while-statement shall be a scalar Boolean expression. The while-statement corresponds to while-statements in programming languages, and is formally defined as follows:

1. The expression of the while-statement is evaluated.
2. If the expression of the while-statement is false, the execution continues after the while-statement.
3. If the expression of the while-statement is true, the entire body of the while-statement is executed (except if a break-statement, see Section 11.2.4, or a return-statement, see Section 11.2.4, is executed), and then execution proceeds at step 1.

11.2.4 Break-Statement

The break-statement breaks the execution of the innermost while or for-loop enclosing the break-statement and continues execution after the while- or for-loop. It can only be used in a while- or for-loop in an algorithm section. It has the following syntax:

```
break;
```

[Example (note this could alternatively use return):

```
function findValue "Returns position of val or 0 if not found"
  input Integer x[:];
  input Integer val;
  output Integer index;
algorithm
  index := size(x,1);
  while index >= 1 loop
    if x[index]== val then
      break;
    else
      index := index - 1;
    end if;
  end while;
end findValue;
]
```

11.2.5 Return-Statements

Can only be used inside functions, see Section 12.1.2.

11.2.6 If-Statement

If-statements have the following syntax:

```
if expression then
  { statement ";" }
{ elseif expression then
  { statement ";" }
}
[ else
  { statement ";" }
]
end if;
```

The expression of an if- or elseif-clause must be scalar Boolean expression. One if-clause, and zero or more elseif-clauses, and an optional else-clause together form a list of branches. One or zero of the bodies of these if-, elseif- and else-clauses is selected, by evaluating the conditions of the if- and elseif-clauses sequentially until a condition that evaluates to true is found. If none of the conditions evaluate to true the body of the else-clause is selected (if an else-clause exists, otherwise no body is selected). In an algorithm section, the selected body is then executed. The bodies that are not selected have no effect on that model evaluation.

11.2.7 When-Statements

A when-statement has the following syntax:

```
when expression then
  { statement ";" }
{ elseif expression then
  { statement ";" } }
end when
```

The expression of a when-statement shall be a discrete-time Boolean scalar or vector expression. The algorithmic statements within a when-statement are activated when the scalar or any one of the elements of the vector-expression becomes true.

[Example:

Algorithms are activated when x becomes > 2 :

```
when x > 2 then
  y1 := sin(x);
  y3 := 2*x + y1+y2;
end when;
```

The statements inside the when-statement are activated when either x becomes > 2 or `sample(0,2)` becomes true or x becomes less than 5:

```
when {x > 2, sample(0,2), x < 5} then
  y1 := sin(x);
  y3 := 2*x + y1+y2;
end when;
```

For when-statements in algorithm sections the order is significant and it is advisable to have only one assignment within the when-statement and instead use several algorithm sections having when-statements with identical conditions, e.g.:

```
algorithm
  when x > 2 then
    y1 := sin(x);
  end when;

equation
  y2 = sin(y1);

algorithm
  when x > 2 then
    y3 := 2*x + y1+y2;
  end when;
```

Merging the when-statements can lead to less efficient code and different models with different behavior depending on the order of the assignment to $y1$ and $y3$ in the algorithm.

]

11.2.7.1 Restrictions on When-Statements

- A when-statement shall not be used within a function.
- When-statements cannot be nested.
- When-statements may not occur inside while, if, and for-clauses in algorithms.

[*Example:*

The following nested when-statement is invalid:

```
when x > 2 then
  when y1 > 3 then
    y2 := sin(x);
  end when;
end when;
]
```

11.2.7.2 Defining When-Statements by If-Statements

A when-statement:

```
algorithm
  when {x>1, ..., y>p} then
    ...
  elseif x > y.start then
    ...
  end when;
```

is similar to the following special if-statement, where Boolean `b1[N]`; and Boolean `b2`; are necessary because the `edge()` operator can only be applied to variables

```
Boolean b1[N] (start={x.start>1, ..., y.start>p});
Boolean b2 (start=x.start>y.start);
algorithm
  b1:={x>1, ..., y>p};
  b2:=x>y.start;

  if edge(b1[1]) or edge(b1[2]) or ... edge(b1[N]) then
    ...
  elseif edge(b2) then
    ...
  end if;
```

with `edge(A) = A` and `not pre(A)` and the additional guarantee, that the statements within this special if-statement are only evaluated at event instants. The difference compared to the when-statements is that e.g. ‘pre’ may only be used on continuous-time real variables inside the body of a when-clause and not inside these if-statements.

11.2.8 Special Statements

These special statements have the same form and semantics as the corresponding equations, apart from the general difference in semantics between equations and statements.

11.2.8.1 Reinit Statement

See Section 8.3.6.

11.2.8.2 Assert Statement

See Section 8.3.7. A failed assert stops the execution of the current algorithm.

11.2.8.3 Terminate Statement

See Section 8.3.8. The terminate statement may not be in functions; In an algorithm outside a function it does not stop the execution of the current algorithm.

Chapter 12

Functions

This chapter describes the Modelica function construct.

12.1 Function Declaration

A Modelica function is a specialized class (Section 12.2) using the keyword `function`. The body of a Modelica function is an algorithm section that contains procedural algorithmic code to be executed when the function is called, or alternatively an external function specifier (Section 12.8). Formal parameters are specified using the `input` keyword, whereas results are denoted using the `output` keyword. This makes the syntax of function definitions quite close to Modelica class definitions, but using the keyword `function` instead of `class`.

[The structure of a typical function declaration is sketched by the following schematic function example:]

```
function functionname
  input  TypeI1 in1;
  input  TypeI2 in2;
  input  TypeI3 in3 := default_expr1 "Comment" annotation(...);
  ...
  output TypeO1 out1;
  output TypeO2 out2 := default_expr2;
  ...
protected
  <local variables>
  ...
algorithm
  ...
  <statements>
  ...
end functionname;
]
```

Optional explicit default values can be associated with any input or output formal parameter through declaration assignments. *[Such defaults are shown for the third input parameter and the second output parameter in our example.]* Comment strings and annotations can be given for any formal parameter declaration, as usual in Modelica declarations.

[All internal parts of a function are optional; i.e., the following is also a legal function:]

```
function functionname
end functionname;
]
```

12.1.1 Ordering of Formal Parameters

The relative ordering between input formal parameter declarations is significant since that determines the matching between actual arguments and formal parameters at function calls with positional parameter passing. Likewise, the relative ordering between the declarations of the outputs is significant since that determines the matching with receiving variables at function calls of functions with multiple results. However, the declarations of the inputs and outputs can be intermixed as long as these internal orderings are preserved. [*Mixing declarations in this way is not recommended, however, since it makes the code hard to read.*]

[*Example:*

```
function <functionname>
  output TypeO1 out1; // Intermixed declarations of inputs and outputs
  input  TypeI1 in1;  // not recommended since code becomes hard to read
  input  TypeI2 in2;
  ...
  output TypeO2 out2;
  input  TypeI3 in3;
  ...
end <functionname>;
```

]

12.1.2 Function Return-Statement

The return-statement terminates the current function call, see Section 12.4. It can only be used in an algorithm section of a function. It has the following form:

```
return;
```

[*Example (note this could alternatively use break):*

```
function findValue "Returns position of val or 0 if not found"
  input Integer x[:];
  input Integer val;
  output Integer index;
algorithm
  for i in 1:size(x,1) loop
    if x[i] == val then
      index := i;
      return;
    end for;
  index := 0;
  return;
end findValue;
```

]

12.1.3 Inheritance of Functions

It is allowed for a function to inherit and/or modify another function following the usual rules for inheritance of classes (Chapter 7). [*For example, it is possible to modify and extend a function class to add default values for input variables.*]

12.2 Function as a Specialized Class

The function concept in Modelica is a specialized class (Section 4.6). [*The syntax and semantics of a function have many similarities to those of the block specialized class. A function has many of the properties of a general class, e.g. being able to inherit other functions, or to redeclare or modify elements of a function declaration.*]

Modelica functions have the following restrictions compared to a general Modelica class:

- Each input formal parameter of the function must be *prefixed* by the keyword `input`, and each result formal parameter by the keyword `output`. All public variables are formal parameters.
- Input formal parameters are read-only after being bound to the actual arguments or default values, i.e., they may not be assigned values in the body of the function.
- A function may *not be used in connections*, may have *no equations*, may have *no initial algorithm*, and can have at most *one algorithm* section, which, if present, is the body of the function.
- A function may have zero or one external function interface, which, if present, is the external definition of the function.
- For a function to be called in a simulation model, it must have either an algorithm section or an external function interface as its body, and it may not be partial.
- A function *cannot contain* calls to the Modelica *built-in operators* `der`, `initial`, `terminal`, `sample`, `pre`, `edge`, `change`, `reinit`, `delay`, `cardinality`, to the operators of the built-in package `Connections`, and is not allowed to contain when-statements.
- The dimension *sizes* not declared with `(:)` of each array result or array local variable [*i.e.*, a *non-input components*] of a function must be either given by the input formal parameters, or given by constant or parameter expressions, or by expressions containing combinations of those (Section 12.4.3).
- The local variables of a function are not automatically initialized to the implicit default values of the data type (Section 12.4.3) [*e.g.* `0.0` for `Real`] *for performance reasons. It is the responsibility of the user to provide explicit defaults or to define the values of such variables before they are referenced.*
- Components of a function will inside the function behave as though they had discrete-time variability.

Modelica functions have the following enhancements compared to a general Modelica `class`:

- A function may be called using the conventional *positional calling syntax* for passing arguments.
- A function can be *recursive*.
- A formal parameter or local variable may be initialized through an *assignment* `(:=)` of a default value in its declaration. Initialization through an equation is not possible.
- A function is dynamically instantiated when it is called rather than being statically instantiated by an instance declaration, which is the case for other kinds of classes.
- A function may have an external function interface specifier as its body.
- A function may have a return statement in its algorithm section body.
- A function allows dimension sizes declared with `(:)` to be resized for non-input array variables, see Section 12.4.4.

12.3 Pure Modelica Functions

Modelica functions are pure, i.e., are side-effect free with respect to the Modelica state (the set of all Modelica variables in a total simulation model), apart from the exceptional case specified further below. This means that:

- Modelica functions are mathematical functions, i.e. calls with the same input argument values always give the same results.
- A Modelica function is side-effect free with respect to the internal Modelica simulation state. Specifically, the ordering of function calls and the number of calls to a function shall not influence the simulation state.

[*Comment 1: This property enables writing declarative specifications using Modelica. It also makes it possible for Modelica compilers to freely perform algebraic manipulation of expressions containing function calls while still preserving their semantics.*]

[*Comment 2: The Modelica translator is responsible for maintaining this property for pure non-external functions. Regarding external functions, the external function implementor is responsible. Note that external functions can have side-effects as long as they do not influence the internal Modelica simulation state, e.g. caching variables for performance or printing trace output to a log file.*]

- Exception: An impure function is either an impure external function or a Modelica function calling an impure function. An impure function (that may return different values at different calls despite having the

same input argument values), may be called from within an impure function, from within a when-equation or when-statement, and during initialization.

[*Comment: The semantics is undefined if the function call is part of an algebraic loop.*

This exceptional case allows, e.g. calling impure external functions at events in when-equations or when-statements during hardware-in-the-loop simulation, to obtain input from the external hardware component, and/or provide output to be communicated to such a component.

The above implies that if all external functions that are used are pure then all Modelica functions are pure.]

12.4 Function Call

Function classes and record constructors can be called as described in this section.

12.4.1 Positional or Named Input Arguments of Functions

A function call has optional positional arguments followed by zero, one or more named arguments, such as

```
f(3.5, 5.76, arg3=5, arg6=8.3);
```

The formal syntax of a function call:

```
primary :
  name function_call_args

function_call_args :
  "(" [ function_arguments ] ")"

function_arguments :
  expression [ "," function_arguments ]
  | named_arguments

named_arguments: named_argument [ "," named_arguments ]

named_argument: IDENT "=" expression
```

The interpretation of a function call is as follows: First, a list of unfilled slots is created for all formal input parameters. If there are N positional arguments, they are placed in the first N slots, where the order of the parameters is given by the order of the component declarations in the function definition. Next, for each named argument `identifier = expression`, the `identifier` is used to determine the corresponding slot. This slot shall be not filled [*otherwise an error occurs*] and the value of the argument is placed in the slot, filling it. When all arguments have been processed, the slots that are still unfilled are filled with the corresponding default value of the function definition. There shall be no remaining unfilled slots [*otherwise an error occurs*] and the list of filled slots is used as the argument list for the call.

The type of each argument must agree with the type of the corresponding parameter, except where the standard type coercions can be used to make the types agree. (See also Section 12.4.5 on applying scalar functions to arrays.)

[*Example.*

Assume a function RealToString is defined as follows to convert a Real number to a String:

```
function RealToString
  input   Real number;
  input   Real precision = 6 "number of significantdigits";
  input   Real length    = 0 "minimum length of field";
  output  String string "number as string";
  ...
end RealToString;
```

Then the following applications are equivalent:

```
RealToString(2.0);
```

```

RealToString(2.0, 6, 0);
RealToString(2.0, 6);
RealToString(2.0, precision=6);
RealToString(2.0, length=0);
RealToString(2.0, 6, precision=6); // error: slot is used twice
]

```

12.4.2 Output Formal Parameters of Functions

A function may have more than one output component, corresponding to multiple return values. The only way to call a function returning more than one result is to make the function call the right hand side of an equation or assignment. In these cases, the left hand side of the equation or assignment shall contain a list of component references within parentheses:

```
(out1, out2, out3) = f(...);
```

The component references are associated with the output components according to their position in the list. Thus output component *i* is set equal to, or assigned to, component reference *i* in the list, where the order of the output components is given by the order of the component declarations in the function definition. The type of each component reference in the list must agree with the type of the corresponding output component.

A function application may be used as expression whose value and type is given by the value and type of the first output component, if exactly one return result is provided.

It is possible to omit left hand side component references and/or truncate the left hand side list in order to discard outputs from a function call.

[Optimizations to avoid computation of unused output results can be automatically deduced by an optimizing compiler].

[Example:

Function "eigen" to compute eigenvalues and optionally eigenvectors may be called in the following ways:

```

      ev = eigen(A);           // calculate eigenvalues
      x  = isStable(eigen(A)); // used in an expression
      (ev, vr) = eigen(A)      // calculate eigenvectors
      (ev,vr,vl) = eigen(A)    // and also left eigenvectors
      (ev,,vl)  = eigen(A)     // no right eigenvectors

```

The function may be defined as:

```

function eigen "calculate eigenvalues and optionally eigenvectors"
  parameter Integer n = size(A,1);
  input      Real    A[:, size(A,1)];
  output    Real    eigenValues[n,2];
  output    Real    rightEigenVectors[n,n];
  output    Real    leftEigenVectors [n,n];
algorithm
  // The output variables are computed separately (and not, e.g., by one
  // call of a Fortran function) in order that an optimizing compiler can remove
  // unnecessary computations, if one or more output arguments are missing
  // compute eigenvalues
  // compute right eigenvectors using the computed eigenvalues
  // compute left eigenvectors using the computed eigenvalues
end eigen;
]

```

The only permissible use of an expression in the form of a list of expressions in parentheses, is when it is used as the left hand side of an equation or assignment where the right hand side is an application of a function.

[Example. The following are illegal:

```

(x+1, 3.0, z/y) = f(1.0, 2.0); // Not a list of component references.
(x, y, z) + (u, v, w)          // Not LHS of suitable eqn/assignment.
]

```

12.4.3 Initialization and Declaration Assignments of Components in Functions

Components in a function can be divided into three groups:

- Public components which are input formal parameters.
- Public components which are output formal parameters.
- Protected components which are local variables, parameters, or constants.

When a function is called components of a function do not have start-attributes. However, a declaration assignment (`:= expression`) with an expression may be present for a component.

A declaration assignment for a non-input component initializes the component to this `expression` at the start of every function invocation (before executing the algorithm section or calling the external function).

Declaration assignments can only be used for components of a function. If no declaration assignment is given for a non-input component its value at the start of the function invocation is undefined. It is a quality of implementation issue to diagnose this for non-external functions. Declaration assignments for input formal parameters are interpreted as default arguments, as described in Section 12.4.1.

[The properties of components in functions described in this section are also briefly described in Section 12.2.]

12.4.4 Flexible Array Sizes and Resizing of Arrays in Functions

[Flexible setting of array dimension sizes of arrays in functions is also briefly described in Section 12.2.]

A dimension size not specified with colon(`:`) for a non-input array component of a function must be given by the inputs or be constant.

[Example:

```
function joinThreeVectors
  input Real v1[:],v2[:],v3[:];
  output Real vres[size(v1,1)+size(v2,1)+size(v3,1)];
algorithm
  vres := cat(1,v1,v2,v3);
end joinThreeVectors;
```

]

Non-input arrays [*i.e., function array result variables or local variables*] declared in functions can be resized according the following rules:

- A non-input array component declared in a function with a dimension size specified by colon(`:`) and no declaration assignment, can change size according to these special rules:
- Prior to execution of the function algorithm the dimension size is zero.
- The entire array (without any subscripts) may be assigned with a corresponding array with arbitrary dimension size (the array variable is re-sized).

These rules also apply if the array component is an element of a record component in a function.

[Example: A function to collect the positive elements in a vector:

```
function collectPositive
  input Real x[:];
  output Real xpos[:];
algorithm
  for i in 1:size(x,1) loop
    if x[i]>0 then
      xpos:=cat(1,xpos,x[i:i]);
    end if;
  end for;
end collectPositive;
```

]

12.4.5 Scalar Functions Applied to Array Arguments

Functions with one scalar return value can be applied to arrays element-wise, e.g. if A is a vector of reals, then $\sin(A)$ is a vector where each element is the result of applying the function \sin to the corresponding element in A . Only function classes that are transitively non-replaceable (Section 6.2.1 7.1.4) may be called vectorized.

Consider the expression $f(\text{arg1}, \dots, \text{argn})$, an application of the function f to the arguments $\text{arg1}, \dots, \text{argn}$ is defined.

For each passed argument, the type of the argument is checked against the type of the corresponding formal parameter of the function.

1. If the types match, nothing is done.
2. If the types do not match, and a type conversion can be applied, it is applied. Continue with step 1.
3. If the types do not match, and no type conversion is applicable, the passed argument type is checked to see if it is an n -dimensional array of the formal parameter type. If it is not, the function call is invalid. If it is, we call this a foreach argument.
4. For all foreach arguments, the number and sizes of dimensions must match. If they do not match, the function call is invalid.
5. If no foreach argument exists, the function is applied in the normal fashion, and the result has the type specified by the function definition.
6. The result of the function call expression is an n -dimensional array with the same dimension sizes as the foreach arguments. Each element $e_{i,\dots,j}$ is the result of applying f to arguments constructed from the original arguments in the following way:
 - If the argument is not a foreach argument, it is used as-is.
 - If the argument is a foreach argument, the element at index $[i,\dots,j]$ is used.

If more than one argument is an array, all of them have to be the same size, and they are traversed in parallel.

[Examples:

```
sin({a, b, c})      = {sin(a), sin(b), sin(c)}    // argument is a vector
sin([a,b,c])        = [sin(a),sin(b),sin(c)]      // argument may be a matrix
atan({a,b,c},{d,e,f}) = {atan(a,d), atan(b,e), atan(c,f)}
```

This works even if the function is declared to take an array as one of its arguments. If `pval` is defined as a function that takes one argument that is a vector of Reals and returns a Real, then it can be used with an actual argument which is a two-dimensional array (a vector of vectors). The result type in this case will be a vector of Real.

```
pval([1,2;3,4]) = [pval([1,2]); pval([3,4])]
sin([1,2;3,4])  = [sin({1,2}); sin({3,4})]
                = [sin(1), sin(2); sin(3), sin(4)]

function Add
  input Real e1, e2;
  output Real sum1;
algorithm
  sum1 := e1 + e2;
end Add;
```

Add(1, [1,2,3]) adds one to each of the elements of the second argument giving the result [2,3,4]. However, it is illegal to write $1 + [1,2,3]$, because the rules for the built-in operators are more restrictive.]

12.4.6 Empty Function Calls

An “empty” function call is a call that does not return any results. [An empty call is of limited use in Modelica since a function call without results does not contribute to the simulation, and is not allowed to have side-effects that influence the simulation state.]

An empty call can occur either as a kind of “null” equation or “null” statement, [e.g. *as in the empty calls to `eigen()` in the example below*]:

```
equation
  Modelica.Math.Matrices.eigen(A); // Empty function call as an equation
algorithm
  Modelica.Math.Matrices.eigen(A); // Empty function call as a statement
]
```

12.5 Built-in Functions

There are basically four groups of built-in functions in Modelica:

- Intrinsic mathematical and conversion functions, see Section 3.7.1.
- Derivative and special operators with function syntax, see Section 3.7.2.
- Event-related operators with function syntax, see Section 3.7.3.
- Built-in array functions, see Section 10.3.

12.6 Record Constructor Functions

Whenever a record is defined, a record constructor function with the same name and in the same scope as the record class is implicitly defined according to the following rules:

The declaration of the record is partially flattened including inheritance, modifications, redeclarations, and expansion of all names referring to declarations outside of the scope of the record to their fully qualified names [*in order to remove potentially conflicting import statements in the record constructor function due to flattening the inheritance tree*].

All record elements [*i.e., components and local class definitions*] of the partially flattened record declaration are used as declarations in the record constructor function with the following exceptions:

- Component declarations which do not allow a modification [*such as `constant Real c=1` or `final parameter Real`*] are declared as protected components in the record constructor function.
- Prefixes (`constant`, `parameter`, `final`, `discrete`, `input`, `output`, ...) of the remaining record components are removed.
- The prefix `input` is added to the public components of the record constructor function.

An instance of the record is declared as output parameter [*using a name, not appearing in the record*] together with a modification. In the modification, all input parameters are used to set the corresponding record variables.

A record constructor can only be called if the referenced record class is found in the global scope, and thus cannot be modified.

[*This allows to construct an instance of a record, with an optional modification, at all places where a function call is allowed. Examples:*

```
record Complex "Complex number"
  Real re "real part";
  Real im "imaginary part";
end Complex;

function add
  input Complex u, v;
  output Complex w(re=u.re + v.re, im=u.im+v.im);
end add;

Complex c1, c2;
equation
  c2 = add(c1, Complex(sin(time), cos(time)));
```

In the following example, a convenient data sheet library of components is built up:


```

package Motors
  record MotorData "Data sheet of a motor"
    parameter Real inertia;
    parameter Real nominalTorque;
    parameter Real maxTorque;
    parameter Real maxSpeed;
  end MotorData;

  model Motor "Motor model" // using the generic MotorData
    MotorData data;
    ...
  equation
    ...
  end Motor;

  record MotorI123 = MotorData( // data of a specific motor
    inertia      = 0.001,
    nominalTorque = 10,
    maxTorque     = 20,
    maxSpeed      = 3600) "Data sheet of motor I123";

  record MotorI145 = MotorData( // data of another specific motor
    inertia      = 0.0015,
    nominalTorque = 15,
    maxTorque     = 22,
    maxSpeed      = 3600) "Data sheet of motor I145";

end Motors

model Robot
  import Motors.*;
  Motor motor1(data = MotorI123()); // just refer to data sheet
  Motor motor2(data = MotorI123(inertia=0.0012));
  // data can still be modified (if no final declaration in record)
  Motor motor3(data = MotorI145());
  ...
end Robot;

```

Example showing most of the situations, which may occur for the implicit record constructor function creation. With the following record definitions

```

package Demo;
  record Record1;
    parameter Real r0 = 0;
  end Record1;

  record Record2
    import Modelica.Math.*;
    extends Record1;
    constant Real c1 = 2.0;
    constant Real c2;
    parameter Integer n1 = 5;
    parameter Integer n2;
    parameter Real r1 "comment";
    parameter Real r2 = sin(c1);
    final parameter Real r3 = cos(r2);
    Real r4;
    Real r5 = 5.0;
    Real r6[n1];
    Real r7[n2];
  end Record2;
end Demo;

```

the following record constructor functions are implicitly defined

```

package Demo;
  function Record1
    input Real r0 = 0;
    output Record1 'result'(r0 = r0);
  end Record1;

```

```

end Record1;

function Record2
  input Real    r0 = 0;
  input Real    c2;
  input Integer n1 := 5;
  input Integer n2;
  input Real    r1 "comment"; // the comment also copied from record
  input Real    r2 := Modelica.Math.sin(c1);
  input Real    r4;
  input Real    r5 = 5.0;
  input Real    r6[n1];
  input Real    r7[n2];
  output Record2 'result' (r0=r0,c2=c2,n1=n1,n2=n2,r1=r1,r2=r2,
                           r4=r4,r5=r5,r6=r6,r7=r7);

protected
  constant Real c1 = 2.0; // referenced from r2
  final parameter Real r3 = Modelica.Math.cos(r2);
end Record2;
end Demo;

```

and can be applied in the following way

```

Demo.Record2 r1 = Demo.Record2(r0=1, c2=2, n1=2, n2=3, r1=1, r2=2,
                               r4=5, r5=5, r6={1,2}, r7={1,2,3});
Demo.Record2 r2 = Demo.Record2(1,2,2,3,1,2,5,5,{1,2},{1,2,3});
parameter Demo.Record2 r3 = Demo.Record2(c2=2, n2=1, r1=1, r4=4,
                                           r6=1:5, r7={1});

```

The above example is only used to show the different variants appearing with prefixes, but it is not very meaningful, because it is simpler to just use a direct modifier.

]

12.7 Declaring Derivatives of Functions

Derivatives of functions can be declared explicitly using the derivative annotation, see Section 12.7.1, whereas a function can be defined as a partial derivative of another function using the der-operator in a short function definition, see Section 12.7.2.

12.7.1 Using the Derivative Annotation

A function declaration can have an annotation `derivative` specifying the derivative function. This can influence simulation time and accuracy and can be applied to both functions written in Modelica and to external functions. A derivative annotation can state that it is only valid under certain restrictions on the input arguments. These restrictions are defined using the following optional attributes: `order` (only a restriction if `order>1`, the default for `order` is 1), `noDerivative`, and `zeroDerivative`. The given derivative-function can only be used to compute the derivative of a function call if these restrictions are satisfied. There may be multiple restrictions on the derivative, in which case they must all be satisfied. The restrictions also imply that some derivatives of some inputs are excluded from the call of the derivative (since they are not necessary). A function may supply multiple derivative functions subject to different restrictions.

[Example:

```

function foo0 annotation(derivative=foo1); end foo0;
function foo1 annotation(derivative(order=2)=foo2); end foo1;
function foo2 end foo2;

```

]

The inputs to the derivative function of `order 1` are constructed as follows:

- First are all inputs to the original function, and after all them we will in order append one derivative for each input containing reals.
- The outputs are constructed by starting with an empty list and then in order appending one derivative for each output containing reals.
- If the Modelica function call is a n th derivative ($n \geq 1$), i.e. this function call has been derived from an $(n-1)$ th derivative, an `annotation(order=n+1)=...` specifies the $(n+1)$ th derivative, and the $(n+1)$ th derivative call is constructed as follows:
- The input arguments are appended with the $(n+1)$ th derivative, which are constructed in order from the n th order derivatives.
- The output arguments are similar to the output argument for the n th derivative, but each output is one higher in derivative order.

[Example: Given the declarations

```
function foo0
...
input Real x;
input Boolean linear;
input ...;
output Real y;
...
annotation(derivative=foo1);
end foo0;

function foo1
...
input Real x;
input Boolean linear;
input ...;
input Real der_x;
...
output Real der_y;
...
annotation(derivative(order=2)=foo2);
end foo1;

function foo2
...
input Real x;
input Boolean linear;
input ...;
input Real der_x;
...;
input Real der_2_x;
...
output Real der_2_y;
...

```

the equation

$$(...,y(t),...)=foo0(...,x(t),b,...);$$

implies that:

$$(...,d\,y(t)/dt,...)=foo1(...,x(t),b,..., \dots,d\,x(t)/dt,...);$$

$$(...,d^2\,y(t)/dt^2,...)=foo2(...,x(t),b,...,d\,x(t)/dt,..., \dots,d^2\,x(t)/dt^2,...);$$

]

An input or output to the function may be any simple type (Real, Boolean, Integer, String and enumeration types) or a record, provided the record does not contain both reals and non-reals predefined types. The function must have at least one input containing reals. The output list of the derivative function may not be empty.

- `zeroDerivative=input_var1`

The derivative function is only valid if `input_var1` is independent of the variables the function call is differentiated with respect to (i.e. that the derivative of `input_var1` is “zero”). The derivative of `input_var1` is excluded from the argument list of the derivative-function.

[Assume that function `f` takes a matrix and a scalar. Since the matrix argument is usually a parameter expression it is then useful to define the function as follows (the additional derivative = `f_general_der` is optional and can be used when the derivative of the matrix is non-zero).

```
function f "Simple table lookup"
  input Real x;
  input Real y[:, 2];
  output Real z;
  annotation(derivative(zeroDerivative=y) = f_der,
             derivative=f_general_der);
algorithm
  ...
end f;

function f_der "Derivative of simple table lookup"
  input Real x;
  input Real y[:, 2];
  input Real x_der;
  output Real z_der;
algorithm
  ...
end f_der;

function f_general_der "Derivative of table lookup taking into account varying tables"
  input Real x;
  input Real y[:, 2];
  input Real x_der;
  input Real y_der[:, 2];
  output Real z_der;
algorithm
  ...
end f_general_der;
]
```

- `noDerivative(input_var2 = f(input_var1, ...))`

The derivative function is only valid if the input argument `input_var2` is computed as `f(input_var1, ...)`. The derivative of `input_var2` is excluded from the argument list of the derivative-function.

[Assume that function `fg` is defined as a composition $f(x, g(x))$. When differentiating `f` it is useful to give the derivative under the assumption that the second argument is defined in this way:

```
function fg
  input Real x;
  output Real z;
algorithm
  z := f(x, g(x));
end fg;

function f
  input Real x;
  input Real y;
  output Real z;
  annotation(derivative(noDerivative(y = g(x))) = f_der);
algorithm
  ...
end f;

function f_der
  input Real x;
  input Real x_der;
  input Real y;
```

```

    output Real z_der;
algorithm
...
end f_der;

```

This is useful if g represents the major computational effort of $f(g)$.

12.7.2 Partial Derivatives of Functions

A class defined as:

```
IDENT "=" der "(" name "," IDENT { "," IDENT } ")" comment
```

is the partial derivative of a function, and may only be used as declarations of functions.

The semantics is that a function [*and only a function*] can be specified in this form, defining that it is the partial derivative of the function to the right of the equal sign (looked up in the same way as a short class definition - the looked up name must be a function), and partially differentiated with respect to each IDENT in order (starting from the first one). The IDENT must be Real inputs to the function.

The comment allows a user to comment the function (in the info-layer and as one-line description, and as icon).

[Example: The specific enthalpy can be computed from a Gibbs-function as follows:

```

function Gibbs
  input Real p,T;
  output Real g;
algorithm
...
end Gibbs;

function Gibbs_T=der(Gibbs, T);

function specificEnthalpy
  input Real p,T;
  output Real h;
algorithm
  h:=Gibbs(p,T)-T*Gibbs_T(p,T);
end specificEnthalpy;
]

```

12.8 External Function Interface

Here, the word function is used to refer to an arbitrary external routine, whether or not the routine has a return value or returns its result via output parameters (or both). The Modelica external function call interface provides the following:

- Support for external functions written in C and FORTRAN 77. Other languages, e.g. C++ and Fortran 90, may be supported in the future.
- Mapping of argument types from Modelica to the target language and back.
- Natural type conversion rules in the sense that there is a mapping from Modelica to standard libraries of the target language.
- Handling arbitrary parameter order for the external function.
- Passing arrays to and from external functions where the dimension sizes are passed as explicit integer parameters.
- Handling of external function parameters which are used both for input and output.

The format of an external function declaration is as follows.

```

function IDENT string_comment
{ component_clause ";" }
[ protected { component_clause ";" } ]

```

```

external [ language_specification ] [ external_function_call ]
[annotation ] ";"
[ annotation ";" ]
end IDENT;

```

Components in the public part of an external function declaration shall be declared either as input or output. *[This is just as for any other function. The components in the protected part allows local variables for temporary storage to be declared.]*

The language_specification must currently be one of "builtin", "C" or "FORTRAN 77". Unless the external language is specified, it is assumed to be "C".

The "builtin" specification is only used for functions that are defined to be built-in in Modelica. The external-function call mechanism for "builtin" functions is implementation-defined.

[Example:

```

package Modelica
  package Math
    function sin
      input Real x;
      output Real y;
      external "builtin";
    end sin;
  end Math;
end Modelica;

model UserModel
  parameter Real p=Modelica.Math.sin(2);
end UserModel;

```

]

The external-function-call specification allows functions whose prototypes do not match the default assumptions as defined below to be called. It also gives the name used to call the external function. If the external call is not given explicitly, this name is assumed to be the same as the Modelica name.

The only permissible kinds of expressions in the argument list are identifiers, scalar constants, and the function size applied to an array and a constant dimension number. The annotations are used to pass additional information to the compiler when necessary.

12.8.1 Argument type Mapping

The arguments of the external function are declared in the same order as in the Modelica declaration, unless specified otherwise in an explicit external function call. Protected variables (i.e. temporaries) are passed in the same way as outputs, whereas constants and size-expression are passed as inputs.

12.8.1.1 Simple Types

Arguments of *simple* types are by default mapped as follows for C:

<i>Modelica</i>	<i>C</i>	
	<i>Input</i>	<i>Output</i>
Real	double	double *
Integer	int	int *
Boolean	int	int *
String	const char *	const char **
Enumeration type	int	int *

An exception is made when the argument is of the form `size(..., ...)`. In this case the corresponding C-type is `size_t`.

Strings are NUL-terminated (i.e., terminated by '\0') to facilitate calling of C functions. When returning a non-literal string, the memory for this string can be allocated with function `ModelicaAllocateString` (see Section 12.8.6) [*It is not suitable to use `malloc`, because a Modelica simulation environment may have its own allocation scheme, e.g., a special stack for local variables of a function*]. After return of the external function, the Modelica environment is responsible for the memory allocated with `ModelicaAllocateString` (e.g., to free this memory, when appropriate). It is not allowed to access memory that was allocated with `ModelicaAllocateString` in a previous call of this external function.

Arguments of simple types are by default mapped as follows for FORTRAN 77:

<i>Modelica</i>	FORTRAN 77	
	<i>Input</i>	<i>Output</i>
Real	DOUBLE PRECISION	DOUBLE PRECISION
Integer	INTEGER	INTEGER
Boolean	LOGICAL	LOGICAL
Enumeration type	INTEGER	INTEGER

Passing strings to FORTRAN 77 subroutines/functions is currently not supported.

Enumeration types used as arguments are mapped to type `int` when calling an external C function, and to type `INTEGER` when calling an external FORTRAN function. The *i*:th enumeration literal is mapped to integer value *i*, starting at one.

Return values are mapped to enumeration types analogously: integer value 1 is mapped to the first enumeration literal, 2 to the second, etc. Returning a value which does not map to an existing enumeration literal for the specified enumeration type is an error.

12.8.1.2 Arrays

Unless an explicit function call is present in the external declaration, an array is passed by its address followed by *n* arguments of type `size_t` with the corresponding array dimension sizes, where *n* is the number of dimensions. [*The type `size_t` is a C unsigned integer type.*]

Arrays are by default stored in row-major order when calling C functions and in column-major order when calling FORTRAN 77 functions. These defaults can be overridden by the array layout annotation. See the example below.

The table below shows the mapping of an array argument in the absence of an explicit external function call when calling a C function. The type *T* is allowed to be any of the simple types which can be passed to C as defined in Section 12.8.1.1 or a record type as defined in Section 12.8.1.3 and it is mapped to the type *T'* as defined in these sections.

<i>Modelica</i>	C
	<i>Input and Output</i>
<i>T</i> [<i>dim</i> ₁]	<i>T'</i> *, <code>size_t dim₁</code>
<i>T</i> [<i>dim</i> ₁ , <i>dim</i> ₂]	<i>T'</i> *, <code>size_t dim₁</code> , <code>size_t dim₂</code>
<i>T</i> [<i>dim</i> ₁ , ..., <i>dim</i> _{<i>n</i>}]	<i>T'</i> *, <code>size_t dim₁</code> , ..., <code>size_t dim_{<i>n</i>}</code>

The method used to pass array arguments to FORTRAN 77 functions in the absence of an explicit external function call is similar to the one defined above for C: first the address of the array, then the dimension sizes as integers. See the table below. The type *T* is allowed to be any of the simple types which can be passed to FORTRAN 77 as defined in Section 12.8.1.1 and it is mapped to the type *T'* as defined in that section.

<i>Modelica</i>	FORTRAN 77
	<i>Input and Output</i>
<i>T</i> [<i>dim</i> ₁]	<i>T'</i> , <code>INTEGER dim₁</code>
<i>T</i> [<i>dim</i> ₁ , <i>dim</i> ₂]	<i>T'</i> , <code>INTEGER dim₁</code> , <code>INTEGER dim₂</code>

$T[\dim_1, \dots, \dim_n]$	$T', \text{INTEGER } \dim_1, \dots, \text{INTEGER } \dim_n$
----------------------------	---

[The following two examples illustrate the default mapping of array arguments to external C and FORTRAN 77 functions.

```
function foo
  input  Real    a[:, :, :];
  output Real    x;
  external;
end foo;
```

The corresponding C prototype is as follows:

```
double foo(double *, size_t, size_t, size_t);
```

If the external function is written in FORTRAN 77, i.e.:

```
function foo
  input  Real    a[:, :, :];
  output Real    x;
  external "FORTRAN 77";
end foo;
```

the default assumptions correspond to a FORTRAN 77 function defined as follows:

```
FUNCTION foo(a, d1, d2, d3)
  DOUBLE PRECISION(d1,d2,d3) a
  INTEGER          d1
  INTEGER          d2
  INTEGER          d3
  DOUBLE PRECISION          foo
  ...
END
```

]

When an explicit call to the external function is present, the array and the sizes of its dimensions must be passed explicitly.

[This example shows how to arrays can be passed explicitly to an external FORTRAN 77 function when the default assumptions are unsuitable.

```
function foo
  input  Real    x[:];
  input  Real    y[size(x,1), :];
  input  Integer i;
  output Real    u1[size(y,1)];
  output Integer u2[size(y,2)];
  external "FORTRAN 77" myfoo(x, y, size(x,1), size(y,2),
                             u1, i, u2);
end foo;
```

The corresponding FORTRAN 77 subroutine would be declared as follows:

```
SUBROUTINE myfoo(x, y, n, m, u1, i, u2)
  DOUBLE PRECISION(n)    x
  DOUBLE PRECISION(n,m)  y
  INTEGER                n
  INTEGER                m
  DOUBLE PRECISION(n)    u1
  INTEGER                i
  DOUBLE PRECISION(m)    u2
  ...
END
```

This example shows how to pass an array in column major order to a C function.

```
function fie
  input Real[:, :] a;
  output Real b;
  external;
```



```

    annotation(arrayLayout = "columnMajor");
end fie;

```

This corresponds to the following C-prototype:

```

double fie(double *, size_t, size_t);
]

```

12.8.1.3 Records

Mapping of record types is only supported for C. A Modelica record class that contains simple types, other record elements, or arrays with fixed dimensions thereof, is mapped as follows:

- The record class is represented by a struct in C.
- Each element of the Modelica record is mapped to its corresponding C representation.
- The elements of the Modelica record class are declared in the same order in the C struct.
- Arrays are mapped to the corresponding C array, taking the default array layout or any explicit arrayLayout-directive into consideration.

Records are passed by reference (i.e. a pointer to the record is being passed).

[For example:

```

record R
  Real x;
  Integer y[10];
  Real z;
end R;

```

is mapped to:

```

struct R {
  double x;
  int    y[10];
  double z;
};
]

```

12.8.2 Return Type Mapping

If there is a single output parameter and no explicit call of the external function, or if there is an explicit external call in the form of an equation, in which case the LHS must be one of the output parameters, the external routine is assumed to be a value-returning function. Mapping of the return type of functions is performed as indicated in the table below. Storage for arrays as return values is allocated by the calling routine, so the dimensions of the returned array are fixed at call time. Otherwise the external function is assumed not to return anything; i.e., it is really a procedure or, in C, a void-function. *[In this case, argument type mapping according to Section 12.8.1 is performed in the absence of any explicit external function call.]*

Return types are by default mapped as follows for C and FORTRAN 77:

<i>Modelica</i>	<i>C</i>	<i>FORTRAN 77</i>
Real	double	DOUBLE PRECISION
Integer	int	INTEGER
Boolean	int	LOGICAL
String	const char*	Not allowed.
$T[dim_1, \dots, dim_n]$	Not allowed.	Not allowed.
Enumeration type	int	INTEGER
Record	See Section 12.8.1.3.	Not allowed.

The element type τ of an array can be any simple type as defined in Section 12.8.1.1 or, for C, a record type as defined in Section 12.8.1.3.

12.8.3 Aliasing

Any potential aliasing in the external function is the responsibility of the tool and not the user. An external function is not allowed to internally change the inputs (even if they are restored before the end of the function).

[Example:

```
function foo
  input Real x;
  input Real y;
  output Real z:=x;
  external "FORTRAN 77" myfoo(x,y,z);
end foo;
```

The following Modelica function:

```
function f
  input Real a;
  output Real b;
algorithm
  b:=foo(a,a);
  b:=foo(b,2*b);
end f;
```

can on most systems be transformed into the following C function:

```
double f(double a) {
  extern void myfoo_(double*,double*,double*);
  double b,temp1,temp2;
  myfoo_(&a,&a,&b);
  temp1=2*b;
  temp2=b;
  myfoo_(&b,&temp1,&temp2);
  return temp2;
}
```

The reason for not allowing the external function to change the inputs is to ensure that inputs can be stored in static memory and to avoid superfluous copying (especially of matrices). If the routine does not satisfy the requirements the interface must copy the input argument to a temporary. This is rare but occurs e.g. in `dormlq` in some Lapack implementations. In those special cases the writer of the external interface have to copy the input to a temporary. If the first input was changed internally in `myfoo` the designer of the interface would have to change the interface function “foo” to:

```
function foo
  input Real x;
  protected Real xtemp:=x; // Temporary used because myfoo changes its input
  public input Real y;
  output Real z;
  external "FORTRAN 77" myfoo(xtemp,y,z);
end foo;
```

Note that we discuss input arguments for Fortran-routines even though Fortran 77 does not formally have input arguments and forbid aliasing between any pair of arguments to a function (Section 15.9.3.6 of X3J3/90.4). For the few (if any) Fortran 77 compilers that strictly follow the standard and are unable to handle aliasing between input variables the tool must transform the first call of `foo` into

```
temp1=a; /* Temporary to avoid aliasing */
myfoo_(&a,&temp1,&b);
```

The use of the function `foo` in Modelica is uninfluenced by these considerations.

]

12.8.4 Annotations for External Libraries and Include Files

The following annotations are useful in the context of calling external functions from Modelica:

- The annotation(`Library="libraryName"`), used by the linker to include the library file where the compiled external function is available.
- The annotation(`Library={"libraryName1", "libraryName2"}`), used by the linker to include the library files where the compiled external function is available and additional libraries used to implement it.
- The annotation(`Include="includeDirective"`), used to include source files, [e.g., *header files*], needed for calling the external function in the code generated by the Modelica compiler.

[*Example: (ExternalLibraries.mo) to show the use of external functions:*

```

model ExternalLibraries
  Real x(start=1.0), y(start=2.0);
equation
  der(x)=-ExternalFunc1(x);
  der(y)=-ExternalFunc2(y);
end ExternalLibraries;

function ExternalFunc1
  input Real x;
  output Real y;
external
  y=ExternalFunc1_ext(x) annotation(Library="ExternalFunc1 ",
                                   Include="#include \"ExternalFunc1_ext.h\"");
end ExternalFunc1;

function ExternalFunc2
  input Real x;
  output Real y;
  external "C" annotation(Library="ExternalFunc2 ",
                         Include="#include \"ExternalFunc2.h\"");
end ExternalFunc2;
]

```

The `LibraryName` is then mapped to a linkage directive in a compiler-dependent way.

12.8.5 Examples

12.8.5.1 Input Parameters, Function Value

[*Here all parameters to the external function are input parameters. One function value is returned. If the external language is not specified, the default is "C", as below.*

```

function foo
  input Real x;
  input Integer y;
  output Real w;
  external;
end foo;

```

This corresponds to the following C-prototype:

```
double foo(double, int);
```

Example call in Modelica:

```
z = foo(2.4, 3);
```

Translated call in C:

```
z = foo(2.4, 3);
```

```
]
```

12.8.5.2 Arbitrary Placement of Output Parameters, No External Function Value

[In the following example, the external function call is given explicitly which allows passing the arguments in a different order than in the Modelica version.

```
function foo
  input Real    x;
  input Integer y;
  output Real   u1;
  output Integer u2;
  external "C" myfoo(x, u1, y, u2);
end foo;
```

This corresponds to the following C-prototype:

```
void myfoo(double, double *, int, int *);
```

Example call in Modelica:

```
(z1,i2) = foo(2.4, 3);
```

Translated call in C:

```
myfoo(2.4, &z1, 3, &i2);
]
```

12.8.5.3 External Function with Both Function Value and Output Variable

[The following external function returns two results: one function value and one output parameter value. Both are mapped to Modelica output parameters.

```
function foo
  input Real    x;
  input Integer y;
  output Real   funcvalue;
  output Integer out1;
  external "C" funcvalue = myfoo(x, y, out1);
end foo;
```

This corresponds to the following C-prototype:

```
double myfoo(double, int, int *);
```

Example call in Modelica:

```
(z1,i2) = foo(2.4, 3);
```

Translated call in C:

```
z1 = myfoo(2.4, 3, &i2);
]
```

12.8.6 Utility Functions

The following utility functions can be called in external Modelica functions written in C. These functions are defined in file `ModelicaUtilities.h`:

ModelicaMessage	<code>void ModelicaMessage(const char* string)</code> Output the message string (no format control).
ModelicaFormatMessage	<code>void ModelicaFormatMessage(const char* string,...)</code> Output the message under the same format control as the C-function <code>printf</code> .
ModelicaError	<code>void ModelicaError(const char* string)</code> Output the error message string (no format control). This function never returns to the calling function, but handles

	the error similarly to an assert in the Modelica code.
ModelicaFormatError	<i>void ModelicaFormatError(const char* string, ...)</i> Output the error message under the same format control as the C-function <code>printf</code> . This function never returns to the calling function, but handles the error similarly to an assert in the Modelica code.
ModelicaAllocateString	<i>char* ModelicaAllocateString(size_t len)</i> Allocate memory for a Modelica string which is used as return argument of an external Modelica function. Note, that the storage for string arrays (= pointer to string array) is still provided by the calling program, as for any other array. If an error occurs, this function does not return, but calls "ModelicaError".
ModelicaAllocateStringWithErrorReturn	<i>char* ModelicaAllocateStringWithErrorReturn(size_t len)</i> Same as <code>ModelicaAllocateString</code> , except that in case of error, the function returns 0. This allows the external function to close files and free other open resources in case of error. After cleaning up resources use <code>ModelicaError</code> or <code>ModelicaFormatError</code> to signal the error.

12.8.7 External Objects

External functions may have internal memory reported between function calls. Within Modelica this memory is defined as instance of the predefined class `ExternalObject` according to the following rules:

- There is a predefined partial class `ExternalObject` [*since the class is partial, it is not possible to define an instance of this class*].
- An external object class shall be directly extended from `ExternalObject`, shall have exactly two function definitions, called "constructor" and "destructor", and shall not contain other elements.
- The constructor function is called exactly once before the first use of the object. For each completely constructed object, the destructor is called exactly once, after the last use of the object, even if an error occurs. The constructor shall have exactly one output argument in which the constructed `ExternalObject` is returned. The destructor shall have no output arguments and the only input argument of the destructor shall be the `ExternalObject`. It is not legal to call explicitly the constructor and destructor functions.
- Classes derived from `ExternalObject` can neither be used in an extends-clause nor in a short class definition.
- External functions may be defined which operate on the internal memory of an `ExternalObject`. An `ExternalObject` used as input argument or return value of an external C-function is mapped to the C-type "void*".

[Example:

A user-defined table may be defined in the following way as an `ExternalObject` (the table is read in a user-defined format from file and has memory for the last used table interval):

```
class MyTable
  extends ExternalObject;
  function constructor
    input String fileName = "";
    input String tableName = "";
    output MyTable table;
    external "C" table = initMyTable(fileName, tableName);
  end constructor;
```

```
function destructor "Release storage of table"
  input MyTable table;
  external "C" closeMyTable(table);
end destructor;
end MyTable;
```

and used in the following way:

```
model test "Define a new table and interpolate in it"
  MyTable table=MyTable(fileName="testTables.txt",
                        tableName="table1"); // call initMyTable

  Real y;
equation
  y = interpolateMyTable(table, time);
end test;
```

This requires to provide the following Modelica function:

```
function interpolateMyTable "Interpolate in table"
  input MyTable table;
  input Real u;
  output Real y;
  external "C" y = interpolateMyTable(table, u);
end interpolateTable;
```

The external C-functions may be defined in the following way:

```
typedef struct { /* User-defined datastructure of the table */
  double* array; /* nrow*ncolumn vector */
  int nrow; /* number of rows */
  int ncol; /* number of columns */
  int type; /* interpolation type */
  int lastIndex; /* last row index for search */
} MyTable;

void* initMyTable(char* fileName, char* tableName) {
  MyTable* table = malloc(sizeof(MyTable));
  if ( table == NULL ) ModelicaError("Not enough memory");
  // read table from file and store all data in *table
  return (void*) table;
};

void closeMyTable(void* object) { /* Release table storage */
  MyTable* table = (MyTable*) object;
  if ( object == NULL ) return;
  free(table->array);
  free(table);
}

double interpolateMyTable(void* object, double u) {
  MyTable* table = (MyTable*) object;
  double y;
  // Interpolate using "table" data (compute y)
  return y;
};
```

]

Chapter 13

Packages

13.1 Package as Specialized Class

The package concept is a specialized class (Section 4.6), using the keyword `package`.

13.2 Motivation and Usage of Packages

[Packages in Modelica may contain definitions of constants and classes including all kinds of specialized classes, functions, and subpackages. By the term subpackage we mean that the package is declared inside another package, no inheritance relationship is implied. Parameters and variables cannot be declared in a package. The definitions in a package should typically be related in some way, which is the main reason they are placed in a particular package. Packages are useful for a number of reasons:

- *Definitions that are related to some particular topic are typically grouped into a package. This makes those definitions easier to find and the code more understandable.*
- *Packages provide encapsulation and coarse-grained structuring that reduces the complexity of large systems. An important example is the use of packages for construction of (hierarchical) class libraries.*
- *Name conflicts between definitions in different packages are eliminated since the package name is implicitly prefixed to names of definitions declared in a package.*
- *Information hiding and encapsulation can be supported to some extent by declaring protected classes, types, and other definitions that are available only inside the package and therefore inaccessible to outside code.*
- *Modelica defines a method for locating a package by providing a standard mapping of package names to storage places, typically file or directory locations in the file system.*

]

13.2.1 Importing Definitions from a Package

The import-clause makes public classes and other public definitions declared in some package available for use by shorter names in a class or a package. It is the only way of referring to definitions declared in some other package for use inside an encapsulated package or class.

[Import-clauses in a package or class fill the following two needs:

- *Making definitions from other packages available for use (by shorter names) in a package or class.*
- *Explicit declaration of usage dependences on other packages.*

]

An import-clause can occur in one of the following five syntactic forms:

```
import packagename ; (qualified import)
```

<code>import [packagename.] definitionname;</code>	(single definition import)
<code>import packagename.*;</code>	(unqualified import)
<code>import shortpackagename = packagename;</code>	(renaming import)
<code>import shortpackagename = [packagename.]definitionname;</code>	(renaming single def. import)

Here *packagename* is the fully qualified name of the imported package including possible dot notation and *definitionname* is the name of an element in a package

13.2.1.1 Lookup of Imported Names

This section only defines how the imported name is looked up in the import clause. For lookup in general – including how import clauses are used, see Section 5.3.

Lookup of the name of an imported package or class, e.g. `A.B.C` in the clauses `import A.B.C;` `import D=A.B.C;` `import A.B.C.*;`, deviates from the normal lexical lookup by starting the lexical lookup of the first part of the name at the top-level.

Qualified import clauses may only refer to packages or elements of packages, i.e., in `import A.B.C;` or `import D=A.B.C;`, `A.B` must be a package. Unqualified import clauses may only import from packages, i.e., in `import A.B.*;`, `A.B` must be a package. [Note: in `import A;` the class `A` can be any class which is an element of the unnamed top-level package]

[For example, if the package `ComplexNumbers` would have been declared as a subpackage inside the package `Modelica.Math`, its fully qualified name would be `Modelica.Math.ComplexNumbers`. Definitionname is the simple name without dot notation of a single definition that is imported. A shortpackagename is a simple name without dot notation that can be used to refer to the package after import instead of the presumably much longer packagename.

The four forms of import are exemplified below assuming that we want to access the addition operation of the hypothetical package `Modelica.Math.ComplexNumbers`:

```
import Modelica.Math.ComplexNumbers;      // Accessed by ComplexNumbers.Add
import Modelica.Math.ComplexNumbers.Add;  // Accessed by Add
import Modelica.Math.ComplexNumbers.*;    // Accessed by Add
import Co = Modelica.Math.ComplexNumbers; // Accessed by Co.Add
]
```

13.2.1.2 Summary of Rules for Import Clauses

The following rules apply to import-clauses:

- Import-clauses are *not* inherited.
- Import-clauses are not named elements of a class or package. This means that import-clauses cannot be changed by modifiers or redeclarations.
- The *order* of import-clauses does not matter.
- One can only import *from* packages, not from other kinds of classes. Both packages and classes can be imported *into* i.e., they may contain import-clauses.
- An imported package or definition should always be referred to by its fully qualified name in the import-clause.
- Multiple qualified import-clauses may not have the same import name.

13.2.2 Mapping Package/Class Structures to a Hierarchical File System

Packages/classes may be represented in the hierarchical structure of the operating system [*the file system or a database*]. For classes with version information see also Section 14.7.3. The nature of such an external entity falls into one of the following two groups:

- Structured entities [*e.g. a directory in the file system*]
- Nonstructured entities [*e.g. a file in the file system*]

13.2.2.1 Mapping a Package/Class Hierarchy into a Directory Hierarchy (Structured Entity)

A structured entity [e.g. *the directory A*] shall contain a node. In a file hierarchy, the node shall be stored in the file `package.mo`. The node shall contain a stored-definition that defines a class [A] with a name matching the name of the structured entity. [The node typically contains documentation and graphical information for a package, but may also contain additional elements of the class A.]

A structured entity may also contain one or more sub-entities (structured or non-structured). The sub-entities are mapped as elements of the class defined by their enclosing structured entity. [For example, if directory A contains the three files `package.mo`, `B.mo` and `C.mo` the classes defined are A, A.B, and A.C.] Two sub-entities shall not define classes with identical names [for example, a directory shall not contain both the sub-directory A and the file A.mo].

13.2.2.2 Mapping a Package/Class Hierarchy into a Single File (Nonstructured Entity)

A nonstructured entity [e.g. *the file A.mo*] shall contain only a model-definition that defines a class [A] with a name matching the name of the nonstructured entity.

13.2.2.3 The within Clause

A within-clause has the following syntax:

```
within [ packageprefixname ] ";"
```

A non-top-level entity shall begin with a within-clause which for the class defined in the entity specifies the location in the Modelica class hierarchy. A top-level class may contain a within-clause with no name.

For a sub-entity of an enclosing structured entity, the within-clause shall designate the class of the enclosing entity.

[Example: The subpackage `Rotational` declared within `Modelica.Mechanics` has the fully qualified name `Modelica.Mechanics.Rotational`, which is formed by concatenating the `packageprefixname` with the short name of the package. The declaration of `Rotational` could be given as below:

```
within Modelica.Mechanics;
package Rotational // Modelica.Mechanics.Rotational
...
]
```

13.2.3 The Modelica Library Path—MODELICAPATH

The top-level scope implicitly contains a number of classes stored externally. If a top-level name is not found at global scope, a Modelica translator shall look up additional classes in an ordered list of library roots, called MODELICAPATH. [The implementation of MODELICAPATH is tool dependent.]

In addition a tool may define an internal list of libraries, since it is in general not advisable for a program installation to modify global environment variables. The version information for a library (as defined in Section 14.7) may also be used during this search to search for a specific version of the library (e.g. if Modelica library version 2.2 is needed and the first directory in MODELICAPATH contain Modelica library version 2.1, whereas the second directory contains Modelica version 2.2, then Modelica library version 2.2 is loaded from the second directory.).

[The first part of the path A.B.C (i.e., A) is located by searching the ordered list of roots in MODELICAPATH. If no root contains A the lookup fails. If A has been found in one of the roots, the rest of the path is located in A; if that fails, the entire lookup fails without searching for A in any of the remaining roots in MODELICAPATH.]

13.2.3.1 Example of Searching MODELICAPATH

[If during lookup a top-level name is not found in the unnamed top-level scope, the search continues in the package hierarchies stored in these directories. Figure 13-1 shows an example `MODELICAPATH = "C:\library;C:\lib1;C:\lib2"`, with three directories containing the roots of the package hierarchies `Modelica`, `MyLib`, and `ComplexNumbers`. The first two are represented as the subdirectories

C:\library\Modelica and C:\lib1\MyLib, whereas the third is stored as the file C:\lib2\ComplexNumbers.mo.

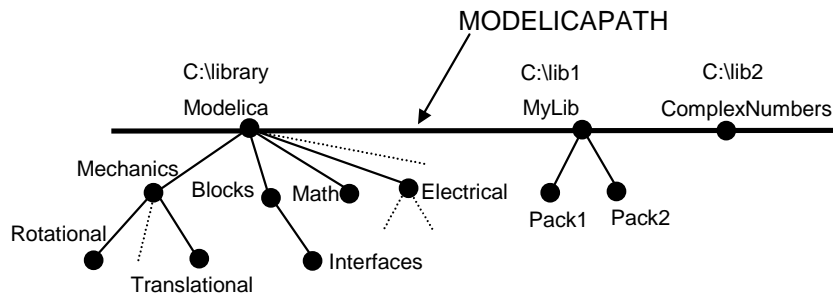


Figure 13-1. Roots of package hierarchies, e.g., Modelica, MyLib, and ComplexNumbers in MODELICAPATH = "C:\library;C:\lib1;C:\lib2".

Assume that we want to access the package `MyLib.Pack2` in Figure 13-1 e.g. through an import clause `import MyLib.Pack2;`. During lookup we first try to find a package `MyLib` corresponding to the first part of the import name. It is not found in the top-level scope since it has not previously been loaded into the environment.

Since the name was not found in the top-level scope the search continues in the directories in the MODELICAPATH in the specified order. For the search to succeed, there must be a subdirectory `MyLib` or a file `MyLib.mo` in one of the directories mentioned in the MODELICAPATH. If there is no such subdirectory or file, the lookup fails. If `MyLib` is found in one of the directories, the rest of the name, in this case `Pack2`, is located in `MyLib`. If that fails, the entire lookup fails without continuing the search in possibly remaining directories.

In this example the name matches the subdirectory named `MyLib` in the second directory "C:\lib1" mentioned in the MODELICAPATH. This subdirectory must have a file `package.mo` containing a definition of the package `MyLib`, according to the Modelica rules on how to map a package hierarchy to the file system. The subpackage `Pack2` is stored in its own subdirectory or file in the subdirectory `MyLib`. In this case the search succeeds and the package `MyLib.Pack2` is loaded into the environment.]

Chapter 14

Annotations

Annotations are intended for storing extra information about a model, such as graphics, documentation or versioning, etc. A Modelica tool is free to define and use other annotations, in addition to those defined here, according to section 14.1. The only requirement is that any tool shall save files with all annotations from this chapter and all vendor-specific annotations intact. To ensure this, annotations must be represented with constructs according to the Modelica grammar. The specification in this document defines the semantic meaning if a tool implements any of these annotations.

14.1 Vendor-Specific Annotations

A vendor may – anywhere inside an annotation – add specific, possibly undocumented, annotations which are not intended to be interpreted by other tools. Two variants of vendor-specific annotations exist; one simple and one hierarchical. Double underscore concatenated with a vendor name as initial characters of the identifier are used to identify vendor-specific annotations.

[Example:

```
annotation (
  Icon(coordinateSystem(extent={{-100,-100}, {100,100}}),
    graphics={__NameOfVendor(Circle(center={0,0}, radius=10))})
);
```

This introduces a new graphical primitive Circle using the hierarchical variant of vendor-specific annotations.

```
annotation (
  Icon(coordinateSystem(extent={{-100,-100}, {100,100}}),
    graphics={Rectangle(extent={{-5,-5}, {7,7}}, __NameOfVendor_shadow=2)})
);
```

This introduces a new attribute __NameOfVendor_shadow for the Rectangle primitive using the simple variant of vendor-specific annotations.]

14.2 Annotations for Documentation

```
documentation_annotation:
  annotation(" Documentation "(" "info" "=" STRING
    ["," "revisions" "=" STRING ] ")" ")"
```

The “Documentation” annotation can contain the “info” annotation giving a textual description, the “revisions” annotation giving a list of revisions and other annotations defined by a tool [The “revisions” documentation may be omitted in printed documentation]. How the tool interprets the information in “Documentation” is unspecified. Within a string of the “Documentation” annotation, the tags <HTML> and

</HTML> or <html> and </html> define optionally begin and end of content that is HTML encoded. Links to Modelica classes may be defined with the HTML link command using scheme “Modelica”, e.g.,

```
<a href="Modelica://MultiBody.Tutorial">MultiBody.Tutorial</a>
```

Together with scheme “Modelica” the (URI) fragment specifiers #diagram, #info, #text, #icon may be used to reference different layers. Example:

```
<a href="Modelica://MultiBody.Joints.Revolute#info">Revolute</a>
```

preferred view_annotation:

```
annotation(" preferredView =" ("info" | "diagram" | "text") ")"
```

The preferredView annotation defines the default view when selecting the class. info means info layer, i.e., the documentation of the class, diagram means diagram layer and text means the Modelica text layer.

14.3 Annotations for Code Generation

code_annotation:

```
annotation(" codeGenerationFlag =" ( false | true ) ")"
```

codeGenerationFlag :

```
"Evaluate" | "HideResult" | "Inline" | "LateInline"
```

These annotations can influence the code generation. The details are defined in the next table:

Evaluate	<p>Has only an effect for a declaration with the prefix parameter.</p> <p>If Evaluate = true, the model developer proposes to utilize the value for the symbolic processing. In that case, it is not possible to change the parameter value after symbolic pre-processing.</p> <p>If Evaluate = false, the model developer proposes to not utilize the value of the corresponding parameter for the symbolic processing.</p> <p><i>[Evaluate is for example used for axis of rotation parameters in the Modelica.Mechanics.MultiBody library in order to improve the efficiency of the generated code]</i></p>
HideResult	<p>HideResult = true defines that the model developer proposes to not show the simulator results of the corresponding component <i>[e.g., it will not be possible to plot this variable]</i>.</p> <p>HideResult = false defines that the developer proposes to show the corresponding component <i>[if a variable is declared in a protected section, a tool might not include it in a simulation result. By setting HideResult = false, the modeler would like to have the variable in the simulation result, even if in the protected section]</i>.</p> <p><i>[HideResult is for example used in the connectors of the Modelica.StateGraph library to not show variables to the modeler that are of no interest to him and would confuse him]</i></p>
Inline	<p>Has only an effect within a function declaration.</p> <p>If “Inline = true”, the model developer proposes to inline the function. This means, that the body of the function is included at all places where the function is called.</p> <p>If “Inline = false”, the model developer proposes to not inline the function.</p> <p><i>[Inline = true is for example used in</i></p>

	Modelica.Mechanics.MultiBody.Frames <i>and in functions of Modelica.Media to have no overhead for function calls such as resolving a vector in a different coordinate system and at the same time the function can be analytically differentiated, e.g., for index reduction needed for mechanical systems.</i>]
LateInline	<p>Has only an effect within a function declaration</p> <p>If “LateInline = true”, the model developer proposes to inline the function after all symbolic transformations have been performed, but before common subexpression elimination takes place.</p> <p>If “LateInline = false”, the model developer proposes to not inline the function after symbolic transformations have been performed.</p> <p>[This annotation is for example used in Modelica.Media.Water.IF97_Uutilities.T_props_ph to provide in combination with common subexpression elimination the automatic caching of function calls. Furthermore, it is used in order that a tool is able to propagate specific enthalpy over connectors in the Modelica_Fluid library.]</p>

```
smoothOrder_annotation:
  annotation "(" smoothOrder "=" UNSIGNED_INTEGER ")"
```

This annotation has only an effect within a function declaration. `smoothOrder` defines the minimum number of differentiations of the function, in order that all of the differentiated outputs are continuous provided all input arguments and their derivatives up to order `smoothOrder` are continuous [This means that the function is at least $C^{\text{smoothOrder}}$. `smoothOrder = 1` means that the function can be differentiated at least once in order that all output arguments are still continuous, provided the input arguments are continuous. If a tool needs the derivative of a function, e.g. for index reduction or to compute an analytic Jacobian, the function can be differentiated analytically at least `smoothOrder` times]

14.4 Annotations for Simulation Experiments

```
experiment_annotation:
  annotation "(" "experiment" "(" [experimentOption] {, experimentOption} ")"

experimentOption:
  StartTime  "=" ["+" | "-"] UNSIGNED_NUMBER |
  StopTime   "=" ["+" | "-"] UNSIGNED_NUMBER |
  Tolerance  "=" UNSIGNED_NUMBER
```

The experiment annotation defines the default start time (`StartTime`) in [s], the default stop time (`StopTime`) in [s], and the default relative integration tolerance (`Tolerance`) for simulation experiments to be carried out with the model or block at hand.

14.5 Annotations for Graphical Objects

A graphical representation of a class consists of two abstraction layers, icon layer and diagram layer showing graphical objects, component icons, connectors and connection lines. The icon representation typically visualizes the component by hiding hierarchical details. The hierarchical decomposition is described in the diagram layer showing icons of subcomponents and connections between these.

Graphical annotations described in this chapter ties into the Modelica grammar as follows.

```
graphical_annotations :
  annotation "(" [ layer_annotations ] ")"
```

```
layer_annotations :
  ( icon_layer | diagram_layer ) ["," layer_annotations ]
```

Layer descriptions (start of syntactic description):

```
icon_layer :
  "Icon" "(" [ coordsys_specification "," ] graphics ")"

diagram_layer :
  "Diagram" "(" [ coordsys_specification "," ] graphics ")"
```

[Example:

```
annotation (
  Icon(coordinateSystem(extent={{-100,-100}, {100,100}}),
    graphics={Rectangle(extent={{-100,-100}, {100,100}}),
      Text({{-100,-100}, {100,100}}, textString="Icon")});
]
```

The graphics is specified as an ordered sequence of graphical primitives, which are described below. *[Note that the ordered sequence is syntactically a valid Modelica annotation, although there is no mechanism for defining an array of heterogeneous objects in Modelica.]*

14.5.1 Common Definitions

The following common definitions are used to define graphical annotations in the later sections.

```
type DrawingUnit = Real(final unit="mm");
type Point       = DrawingUnit[2] "{x, y}";
type Extent      = Point[2]
  "Defines a rectangular area {{x1, y1}, {x2, y2}}";
```

The interpretation of "unit" is with respect to printer output in natural size (not zoomed).

All graphical entities have a visible attribute which indicates if the entity should be shown.

```
partial record GraphicItem
  Boolean visible = true;
  Point origin = {0, 0};
  Real rotation(quantity="angle", unit="deg")=0;
end GraphicItem;
```

The origin attribute specifies the origin of the graphical item in the coordinate system of the layer in which it is defined. The origin is used to define the geometric information of the item and for all transformations applied to the item. All geometric information is given relative the origin attribute, which by default is {0, 0}.

The rotation attribute specifies the rotation of the graphical item around the point defined by the origin attribute.

14.5.1.1 Coordinate Systems

Each of the layers has its own coordinate system. A coordinate system is defined by the coordinates of two points, the lower left corner and the upper right corner, where the coordinates of the first point shall be less than the coordinates of the second point [*a first quadrant coordinate system*].

The attribute preserveAspectRatio specifies a constraint on the shape of components of the class. If preserveAspectRatio is true, changing the extent of components shall preserve the aspect ratio of the coordinate system of the class.

The attribute initialScale specifies the default component size as initialScale times the size of the of the coordinate system of the class. An application may use a different default value of initialScale.

The attribute grid specifies the spacing between grid points which can be used by tools for alignment of points in the coordinate system [e.g. “snap-to-grid”]. Its use and default value is tool-dependent.

```
record CoordinateSystem
    Extent extent;
    Boolean preserveAspectRatio=true;
    Real initialScale = 0.1;
    DrawingUnit grid[2];
end CoordinateSystem;
```

[Example: A coordinate system for an icon could for example be defined as:

```
CoordinateSystem(extent = {{-10, -10}, {10, 10}});
```

i.e. a coordinate system with width 20 units and height 20 units.]

The coordinate systems for the icon and diagram layers are by default defined as follows; where the array of GraphicsItem represents an ordered list of graphical primitives.

```
record Icon "Representation of the icon layer"
    CoordinateSystem coordinateSystem(extent = {{-100, -100}, {100, 100}});
    GraphicItem[:] graphics;
end Icon;

record Diagram "Representation of the diagram layer"
    CoordinateSystem coordinateSystem(extent = {{-100, -100}, {100, 100}});
    GraphicItem[:] graphics;
end Diagram;
```

14.5.1.2 Graphical Properties

Properties of graphical objects and connection lines are described using the following attribute types.

```
type Color = Integer[3] (min=0, max=255) "RGB representation";

constant Color Black = zeros(3);

type LinePattern = enumeration(None, Solid, Dash, Dot, DashDot, DashDotDot);

type FillPattern = enumeration(None, Solid, Horizontal, Vertical,
    Cross, Forward, Backward, CrossDiag, HorizontalCylinder,
    VerticalCylinder, Sphere);

type BorderPattern = enumeration(None, Raised, Sunken, Engraved);

type Smooth = enumeration(None, Bezier);
```

The FillPattern attributes Horizontal, Vertical, Cross, Forward, Backward and CrossDiag specify fill patterns drawn with the line color over the fill color.

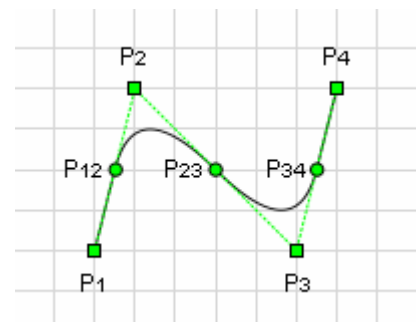
The attributes HorizontalCylinder, VerticalCylinder and Sphere specify gradients that represent a horizontal cylinder, a vertical cylinder and a sphere, respectively. The gradient goes from line color to fill color.

The border pattern attributes Raised, Sunken and Engraved represent frames which are rendered in a tool-dependent way.

The smooth attribute specifies that a line can be drawn as straight line segments (None) or using a spline (Bezier), where the line's points specify control points of a quadratic Bezier curve.

For lines with only two points, the smooth attribute has no effect.

For lines with three or more points (P_1, P_2, \dots, P_n), the middle point of each line segment ($P_{12}, P_{23}, \dots, P_{(n-1)n}$) becomes the starting point and ending points of each quadratic Bezier curve. For each quadratic Bezier curve, the common point of the two line segment becomes the control point. For instance, point P_2 becomes the control point for the Bezier curve starting at P_{12} and ending at



P_{23} . A straight line is drawn between the starting point of the line and the starting point of the first quadratic Bezier curve, as well as between the ending point of the line and the ending point of the last quadratic Bezier curve.

In the illustration above, the square points (P_1 , P_2 , P_3 , and P_4) represent the points that define the line, and the circle points (P_{12} , P_{23} , and P_{34}) are the calculated middle points of each line segment. Points P_{12} , P_2 , and P_{23} define the first quadratic Bezier curve, and the points P_{23} , P_3 , and P_{34} define the second quadratic Bezier curve. Finally a straight line is drawn between points P_1 and P_{12} as well as between P_{34} and P_4 .

```
type Arrow = enumeration(None, Open, Filled, Half);
type TextStyle = enumeration(Bold, Italic, UnderLine);
type TextAlignment = enumeration(Left, Center, Right);
```

Filled shapes have the following attributes for the border and interior.

```
record FilledShape "Style attributes for filled shapes"
  Color lineColor = Black "Color of border line";
  Color fillColor = Black "Interior fill color";
  LinePattern pattern = LinePattern.Solid "Border line pattern";
  FillPattern fillPattern = FillPattern.None "Interior fill pattern";
  DrawingUnit lineThickness = 0.25 "Line thickness";
end FilledShape;
```

14.5.2 Component Instance

A component instance can be placed within a diagram or icon layer. It has an annotation with a Placement modifier to describe the placement. Placements are defined in term of coordinate systems transformations:

```
record Transformation
  Point origin = {0, 0};
  Extent extent;
  Real rotation(quantity="angle", unit="deg")=0;
end Transformation;
```

The origin attribute defines the position of the component in the coordinate system of the enclosing class. The extent defines the position, size and flipping of the component, relative to the origin attribute. The extent is defined relative to the origin attribute of the component instance. Given an extent $\{\{x_1, y_1\}, \{x_2, y_2\}\}$, $x_2 < x_1$ defines horizontal flipping and $y_2 < y_1$ defines vertical flipping around the point defined by the origin attribute.

The rotation attribute specifies rotation of the extent around the point defined by the origin attribute.

The graphical operations are applied in the order: scaling, flipping and rotation.

```
record Placement
  Boolean visible = true;
  Transformation transformation "Placement in the diagram layer";
  Transformation iconTransformation "Placement in the icon layer";
end Placement;
```

[A connector can be shown in both an icon layer and a diagram layer of a class. Since the coordinate systems typically are different, placement information needs to be given using two different coordinate systems. More flexibility than just using scaling and translation is needed since the abstraction views might need different visual placement of the connectors. The attribute transformation gives the placement in the diagram layer and iconTransformation gives the placement in the icon layer. When a connector is shown in a diagram layer, its diagram layer is shown to facilitate opening up a hierarchical connector to allow connections to its internal subconnectors.]

For connectors, the icon layer is used to represent a connector when it is shown in the icon layer of the enclosing model. The diagram layer of the connector is used to represent it when shown in the diagram layer of the enclosing model. Protected connectors are only shown in the diagram layer. Public connectors are shown in both the diagram layer and the icon layer. Non-connector components are only shown in the diagram layer.

14.5.3 Extends clause

Each extends clause may have layer specific annotations which describe the rendering of the base class' icon and diagram layers in the subclass.

```
record IconMap
  Extent extent = {{0, 0}, {0, 0}};
  Boolean primitivesVisible = true;
end IconMap;

record DiagramMap
  Extent extent = {{0, 0}, {0, 0}};
  Boolean primitivesVisible = true;
end DiagramMap;
```

All graphical primitives are inherited from a base class. If the `primitivesVisible` attribute is false, components and connections are by default visible but graphical primitives are not. The base class coordinate system is mapped to the region specified by the attribute extent. If the extent defines a null region (the default), the base class coordinate system shall be fitted to the derived class coordinate system with center alignment and respected `preserveAspectRatio` of the base class.

[Example:

```
model A
  extends B annotation(
    IconMap(extent={{-100,-100}, {100,100}},primitivesVisible=false),
    DiagramMap(extent={{-50,-50}, {0,0}},primitivesVisible=true)
  );
end A;
]
```

14.5.4 Connections

A connection is specified with an annotation containing a `Line` primitive, as specified below. [Example:

```
connect(a.x, b.x)
  annotation(Line(points={{-25,30}, {10,30}, {10, -20}, {40,-20}}));
]
```

14.5.5 Graphical primitives

This section describes the graphical primitives that can be used to define the graphical objects in an annotation.

14.5.5.1 Line

A line is specified as follows:

```
record Line
  extends GraphicItem;
  Point points[:];
  Color color = Black;
  LinePattern pattern = LinePattern.Solid;
  DrawingUnit thickness = 0.25;

  Arrow arrow[2] = {Arrow.None, Arrow.None};    "{start arrow, end arrow}"
  DrawingUnit arrowSize=3;
  Smooth smooth = Smooth.None                    "Spline";
end Line;
```

Note that the `Line` primitive is also used to specify the graphical representation of a connection.

14.5.5.2 Polygon

A polygon is specified as follows:

```
record Polygon
  extends GraphicItem;
  extends FilledShape;
  Point points[:];
  Smooth smooth = Smooth.None           "Spline outline";
end Polygon;
```

The polygon is automatically closed, if the first and the last points are not identical.

14.5.5.3 Rectangle

A rectangle is specified as follows:

```
record Rectangle
  extends GraphicItem;
  extends FilledShape;
  BorderPattern borderPattern = BorderPattern.None;
  Extent extent;
  DrawingUnit radius = 0 "Corner radius";
end Rectangle;
```

The `extent` attribute specifies the bounding box of the rectangle. If the `radius` attribute is specified, the rectangle is drawn with rounded corners of the given radius.

14.5.5.4 Ellipse

An ellipse is specified as follows:

```
record Ellipse
  extends GraphicItem;
  extends FilledShape;
  Extent extent;
  Real startAngle(quantity="angle", unit="deg")=0;
  Real endAngle(quantity="angle", unit="deg")=360;
end Ellipse;
```

The `extent` attribute specifies the bounding box of the ellipse.

For elliptic arcs, `startAngle` and `endAngle` specify the arc prior to the stretch and rotate operations. The arc is drawn counter-clockwise from `startAngle` to `endAngle`, where `startAngle` and `endAngle` are defined counter-clockwise from “3 o’clock”.

14.5.5.5 Text

A text string is specified as follows:

```
record Text
  extends GraphicItem;
  extends FilledShape;
  Extent extent;
  String textString;
  Real fontSize = 0 "unit pt";
  String fontName;
  TextStyle textStyle[:];
  TextAlignment horizontalAlignment = TextAlignment.Center;
end Text;
```

The style attribute `fontSize` specifies the font size. If the `fontSize` attribute is 0 the text is scaled to fit its extent. Otherwise, the size specifies the absolute size. The text is vertically centered in the extent.

If the string `fontName` is empty, the tool may choose a font.

The style attribute `textStyle` specifies variations of the font.

14.5.5.6 Bitmap

A bitmap image is specified as follows:

```
record Bitmap
  extends GraphicItem;
  Extent extent;
  String fileName "Name of bitmap file";
  String imageSource "Base64 representation of bitmap";
end Bitmap;
```

The Bitmap primitive renders a graphical bitmap image. The data of the image can either be stored on an external file or in the annotation itself. The image is scaled to fit the extent.

When the attribute `fileName` is specified, the string refers to an external file containing image data. The mapping from the string to the file is unspecified [the specification should define a portable filename/path format, which can be used here]. The supported file formats include PNG, BMP and JPEG, other supported file formats are unspecified.

When the attribute `imageSource` is specified, the string contains the image data. The image is represented as a Base64 encoding of the image file format (see RFC 4648, <http://tools.ietf.org/html/rfc4648>).

The image is uniformly scaled [*to preserve aspect ratio*] so it exactly fits within the extent [*touching the extent along one axis*]. The center of the image is positioned at the center of the extent.

14.5.6 Variable Graphics and Schematic Animation

Any value (coordinates, color, text, etc) in graphical annotations can be dependent on class variables using the `DynamicSelect` expression. `DynamicSelect` has the syntax of a function call with two arguments, where the first argument specifies the value of the editing state and the second argument the value of the non-editing state. The first argument must be a literal expression. The second argument may contain references to variables to enable a dynamic behavior.

[Example: The level of a tank could be animated by a rectangle expanding in vertical direction and its color depending on a variable overflow:

```
annotation (
  Icon(graphics={Rectangle(
    extent=DynamicSelect({{0,0},{20,20}},{{0,0},{20,level}}),
    fillColor=DynamicSelect({0,0,255},
                           if overflow then {255,0,0} else {0,0,255}))}
);
]
```

14.5.7 User input

It is possible to interactively modify variables during a simulation. The variables may either be parameters, discrete variables or states. New numeric values can be given, a mouse click can change a Boolean variable or a mouse movement can change a Real variable. Input fields may be associated with a `GraphicItem` or a component as an array named `interaction`. The interaction array may occur as an attribute of a graphic primitive, an attribute of a component annotation or as an attribute of the layer annotation of a class.

14.5.7.1 Mouse input

A Boolean variable can be changed when the cursor is held over a graphical item or component and the selection button is pressed if the interaction annotation contains `OnMouseDownSetBoolean`:

```
record OnMouseDownSetBoolean
  Boolean variable "Name of variable to change when mouse button pressed";
  Boolean value "Assigned value";
end OnMouseDown;
```

[Example: A button can be represented by a rectangle changing color depending on a Boolean variable `on` and toggles the variable when the rectangle is clicked on:

```
annotation (Icon(graphics={Rectangle(extent=[0,0; 20,20]),
  fillColor=if on then [255,0,0] else [0,0,255],
  interaction={OnMouseDown(on, not on)}}));
```

```
]
```

In a similar way, a variable can be changed when the mouse button is *released*:

```
record OnMouseUpSetBoolean
  Boolean variable      "Name of variable to change when mouse button released";
  Boolean value         "Assigned value";
end OnMouseClicked;
```

Note that several interaction objects can be associated with the same graphical item or component.

[Example:

```
  interaction={OnMouseDown(on, true), OnMouseUp(on, false)};
```

```
]
```

The `OnMouseMoveXSetReal` interaction object sets the variable to the position of the cursor in X direction in the local coordinate system mapped to the interval defined by the `minValue` and `maxValue` attributes.

```
record OnMouseMoveXSetReal
  Real xVariable        "Name of variable to change when cursor moved
                        in x direction";
  Real minValue;
  Real maxValue;
end OnMouseMoveXSetReal;
```

The `OnMouseMoveYSetReal` interaction object works in a corresponding way as the `OnMouseMoveXSetReal` object but in the Y direction.

```
record OnMouseMoveYSetReal
  Real yVariable        "Name of variable to change when cursor moved
                        in y direction";
  Real minValue;
  Real maxValue;
end OnMouseMoveYSetReal;
```

14.5.7.2 Edit input

The `OnMouseDownEditInteger` interaction object presents an input field when the graphical item or component is clicked on. The field shows the actual value of the variable and allows changing the value. If a too small or too large value according to the min and max parameter values of the variable is given, the input is rejected.

```
record OnMouseDownEditInteger
  Integer variable      "Name of variable to change";
end OnMouseDownEditInteger;
```

The `OnMouseDownEditReal` interaction object presents an input field when the graphical item or component is clicked on. The field shows the actual value of the variable and allows changing the value. If a too small or too large value according to the min and max parameter values of the variable is given, the input is rejected.

```
record OnMouseDownEditReal
  Real variable         "Name of variable to change";
end OnMouseDownEditReal;
```

The `OnMouseDownEditString` interaction object presents an input field when the graphical item or component is clicked on. The field shows the actual value of the variable and allows changing the value.

```
record OnMouseDownEditString
  String variable      "Name of variable to change";
end OnMouseDownEditString;
```

14.6 Annotations for the Graphical User Interface

A class may have the following annotations to define properties of the graphical user interface:

```
annotation(defaultComponentName = "name")
```

When creating a component of the given class, the recommended component name is name.

```
annotation(defaultComponentPrefixes = "prefixes")
```

When creating a component, it is recommended to generate a declaration of the form

```
prefixes class-name component-name
```

The following prefixes may be included in the string prefixes: inner, outer, replaceable, constant, parameter, discrete. *[In combination with `defaultComponentName` it can be used to make it easy for users to create inner components matching the outer declarations; see also example below]*

```
annotation(missingInnerMessage = "message")
```

When an outer component of the class does not have a corresponding inner component, the string message may be used as a diagnostic message.

[Example:

```
model World
  annotation(defaultComponentName      = "world",
             defaultComponentPrefixes = "inner replaceable",
             missingInnerMessage       = "The World object is missing");
  ...
end World;
```

When an instance of model `World` is dragged in to the diagram layer, the following declaration is generated:

```
inner replaceable World world;
]
```

A declaration may have the following annotations:

```
annotation(unassignedMessage = "message");
```

When the variable to which this annotation is attached in the declaration cannot be computed due to the structure of the equations, the string message can be used as a diagnostic message. *[When using BLT partitioning, this means if a variable “a” or one of its aliases “b = a”, “b = -a”, cannot be assigned, the message is displayed. This annotation is used to provide library specific error messages.]*

```
annotation(Dialog(enable = parameter-expression, tab = "tab", group = "group"));
```

Defines the placement of the component or class parameter in a parameter dialog with optional tab and group specification. If enable is false, the input field may be disabled *[and no input can be given]*. “Dialog” is defined as:

```
record Dialog
  parameter String tab    = "General";
  parameter String group  = "Parameters";
  parameter Boolean enable = true;
end Dialog;
```

A parameter dialog is a sequence of tabs with a sequence of groups inside them.

[Examples:

```
connector Frame "Frame of a mechanical system"
...
flow Modelica.SIunits.Force f[3] annotation(unassignedMessage =
  "All Forces cannot be uniquely calculated. The reason could be that the
  mechanism contains a planar loop or that joints constrain the same motion.
  For planar loops, use in one revolute joint per loop the option
  PlanarCutJoint=true in the Advanced menu.
  ");
end Frame;

model BodyShape
...
parameter Boolean animation = true;
parameter SI.Length length "Length of shape"
  annotation(Dialog(enable = animation, tab = "Animation",
    group = "Shape definition"));
...
end BodyShape;
]
```

14.7 Annotations for Version Handling

A top-level package or model can specify the version of top-level classes it uses, its own version number, and if possible how to convert from previous versions. This can be used by a tool to guarantee that consistent versions are used, and if possible to upgrade usage from an earlier version to a current one.

14.7.1 Version Numbering

Version numbers are of the forms:

- Main release versions: `"" UNSIGNED_INTEGER { "." UNSIGNED_INTEGER } ""` [Example: "2.1"]
- Pre-release versions: `"" UNSIGNED_INTEGER { "." UNSIGNED_INTEGER } " " {S-CHAR} ""` [Example: "2.1 Beta 1"]
- Un-ordered versions: `"" NON-DIGIT {S-CHAR} ""` [Example: "Test 1"]

The main release versions are ordered using the hierarchical numerical names, and follow the corresponding pre-release versions. The pre-release versions of the same main release version are internally ordered alphabetically.

14.7.2 Version Handling

In a top-level class, the version number and the dependency to earlier versions of this class are defined using one or more of the following annotations:

- `version = CURRENT-VERSION-NUMBER`
Defines the version number of the model or package. All classes within this top-level class have this version number.
- `conversion (noneFromVersion = VERSION-NUMBER)`
Defines that user models using the VERSION-NUMBER can be upgraded to the CURRENT-VERSION-NUMBER of the current class without any changes.
- `conversion (from (version = VERSION-NUMBER, script = "..."))`
Defines that user models using the VERSION-NUMBER can be upgraded to the CURRENT-VERSION-NUMBER of the current class by applying the given script. The semantics of the conversion script is not defined.
- `uses(IDENT (version = VERSION-NUMBER))`
Defines that classes within this top-level class uses version VERSION-NUMBER of classes within the top-level class IDENT.

The annotations uses and conversion may contain several different sub-entries.

[*Example:*

```
package Modelica
  annotation(version="2.1",
             conversion(noneFromVersion="2.1 Beta 1",
                       from(version="1.5",
                           script="convertFromModelica1_5.mos")));
  ...
end Modelica;

model A
  annotation(version="1.0",
             uses(Modelica(version="1.5")));
  ...
end A;

model B
  annotation(uses(Modelica(version="2.1 Beta 1")));
  ...
end B;
```

In this example the model A uses an older version of the Modelica library and can be upgraded using the given script, and model B uses an older version of the Modelica library but no changes are required when upgrading.

]

14.7.3 Mapping of Versions to File System

A top-level class, IDENT, with version VERSION-NUMBER can be stored in one of the following ways in a directory given in the MODELICAPATH:

- The file IDENT ".mo" [*Example:* Modelica.mo]
- The file IDENT " " VERSION-NUMBER ".mo" [*Example:* Modelica 2.1.mo]
- The directory IDENT [*Example:* Modelica]
- The directory IDENT " " VERSION-NUMBER [*Example:* Modelica 2.1]

This allows a tool to access multiple versions of the same package.

14.8 Annotations for Functions

14.8.1 Function Derivative Annotations

See Section 12.7.1

14.8.2 External Function Annotations

See Section 12.8.4.

14.9 Annotation Choices for Modifications and Redeclarations

See Section 7.3.4.

Chapter 15

Unit Expressions

Unless otherwise stated, the syntax and semantics of unit expressions in Modelica are conform with the international standards ISO 31/0-1992 "General principles concerning quantities, units and symbols" and ISO 1000-1992 "SI units and recommendations for the use of their multiples and of certain other units". Unfortunately, neither these two standards nor other existing or emerging ISO standards define a formal syntax for unit expressions. There are recommendations and Modelica exploits them.

Examples for the syntax of unit expressions used in Modelica: "N.m", "kg.m/s²", "kg.m.s⁻²" "1/rad", "mm/s".

15.1 The Syntax of Unit Expressions

```
unit_expression:
    unit_numerator [ "/" unit_denominator ]

unit_numerator:
    "1" | unit_factors | "(" unit_expression ")"

unit_denominator:
    unit_factor | "(" unit_expression ")"
```

The unit of measure of a dimension free quantity is denoted by "1". The ISO standard does not define any precedence between multiplications and divisions. The ISO recommendation is to have at most one division, where the expression to the right of "/" either contains no multiplications or is enclosed within parentheses. It is also possible to use negative exponents, for example, "J/(kg.K)" may be written as "J.kg⁻¹.K⁻¹".

```
unit_factors:
    unit_factor [ unit_mulop unit_factors ]

unit_mulop:
    "."
```

The ISO standard allows that a multiplication operator symbol is left out. However, Modelica enforces the ISO recommendation that each multiplication operator is explicitly written out in formal specifications. For example, Modelica does not support "Nm" for newtonmeter, but requires it to be written as "N.m".

The preferred ISO symbol for the multiplication operator is a "dot" a bit above the base line: ".". Modelica supports the ISO alternative ".", which is an ordinary "dot" on the base line.

```
unit_factor:
    unit_operand [ unit_exponent ]

unit_exponent:
    [ "+" | "-" ] integer
```

The ISO standard does not define any operator symbol for exponentiation. A `unit_factor` consists of a `unit_operand` possibly suffixed by a possibly signed integer number, which is interpreted as an exponent. There must be no spacing between the `unit_operand` and a possible `unit_exponent`.

```
unit_operand:
    unit_symbol | unit_prefix unit_symbol
```

`unit_prefix:`

`Y | Z | E | P | T | G | M | k | h | da | d | c | m | u | p | f | a | z | y`

A `unit_symbol` is a string of letters. A basic support of units in Modelica should know the basic and derived units of the SI system. It is possible to support user defined unit symbols. In the base version Greek letters is not supported, but full names must then be written, for example "Ohm".

A `unit_operand` should first be interpreted as a `unit_symbol` and only if not successful the second alternative assuming a prefixed operand should be exploited. There must be no spacing between the `unit_symbol` and a possible `unit_prefix`. The value of the prefixes are according to the ISO standard. The letter "u" is used as a symbol for the prefix micro.

15.2 Examples

The unit expression "m" means meter and not milli (10^{-3}), since prefixes cannot be used in isolation. For millimeter use "mm" and for squaremeter, m^2 , write "m2".

The expression "mm2" means $mm^2 = (10^{-3}m)^2 = 10^{-6}m^2$. Note that exponentiation includes the prefix.

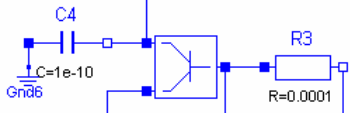
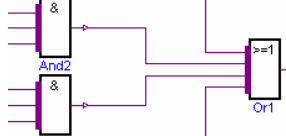
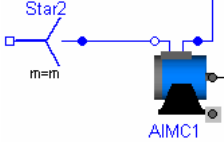
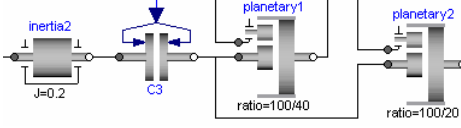
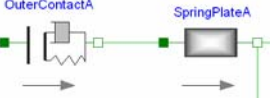
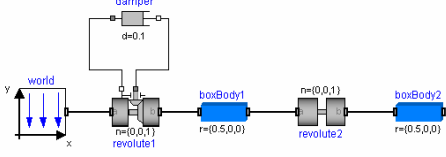
The unit expression "T" means Tesla, but note that the letter "T" is also the symbol for the prefix `tera` which has a multiplier value of 10^{12} .

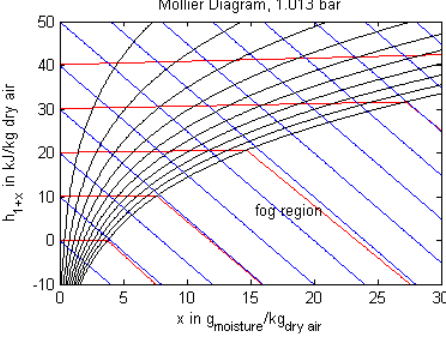
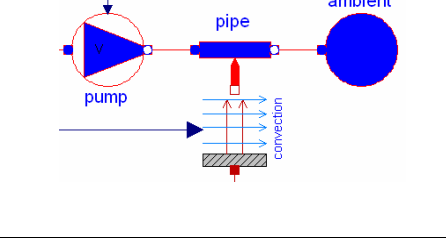
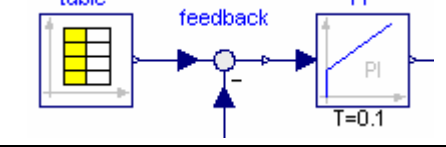
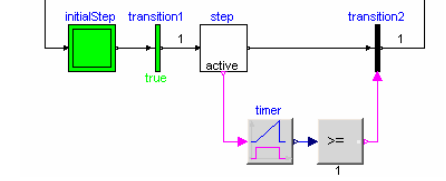


Chapter 16

The Modelica Standard Library






























In order that a modeler can quickly build up system models, it is important that libraries of the most commonly used components are available, ready to use, and sharable between applications. For this reason, the Modelica Association develops and maintains a growing *Modelica Standard Library* called package `Modelica`. This is a free library that can be used without essential restrictions, e.g., in commercial Modelica simulation environments. Furthermore, other people and organizations are developing free and commercial Modelica libraries. For information about these libraries and for downloading the free libraries see www.modelica.org/libraries/.

Version 2.2.2 of the Modelica standard library from Sept. 2007 contains about 740 models and blocks, and 540 functions in the following sublibraries:

	Modelica.Electrical.Analog Analog electrical and electronic components such as resistor, capacitor, transformers, diodes, transistors, transmission lines, switches, sources, sensors
	Modelica.Electrical.Digital Digital electrical components based on VHDL with nine valued logic. Contains delays, gates, sources, and converters between 2-, 3-, 4-, and 9-valued logic.
	Modelica.Electrical.Machines Uncontrolled, electrical machines, such as asynchronous, synchronous and direct current motors and generators.
	Modelica.Mechanics.Rotational 1-dim. rotational mechanical systems, such as drive trains, planetary gear. Contains inertia, spring, gear box, bearing friction, clutch, brake, backlash, torque, etc.
	Modelica.Mechanics.Translational 1-dim. translational mechanical systems, such as mass, stop, spring, backlash, force
	Modelica.Mechanics.MultiBody 3-dim. mechanical systems consisting of joints, bodies, force and sensor elements. Joints can be driven by elements of the Rotational library. Every element has a default animation.

	<p>Modelica.Media</p> <p>Large media library for single and multiple substance fluids with one and multiple phases:</p> <ul style="list-style-type: none"> • High precision gas models based on the NASA Glenn coefficients + mixtures between these gas models • Simple and high precision water models (IAPWS/IF97) • Dry and moist air models • Table based incompressible media. • Simple liquid models with linear compressibility
	<p>Modelica.Thermal</p> <p>Simple thermo-fluid pipe flow, especially for machine cooling systems with water or air fluid. Contains pipes, pumps, valves, sensors, sources etc.</p> <p>Furthermore, lumped heat transfer components are present, such as heat capacitor, thermal conductor, convection, body radiation, etc.</p>
	<p>Modelica.Blocks</p> <p>Continuous and discrete input/output blocks. Contains transfer functions, linear state space systems, non-linear, mathematical, logical, table, source blocks.</p>
	<p>Modelica.StateGraph</p> <p>Hierarchical state machines with similar modeling power as Statecharts. Modelica is used as synchronous “action” language. Deterministic behavior is guaranteed.</p>
<pre>import Modelica.Math.Matrices; A = [1,2,3; 3,4,5; 2,1,4]; b = {10,22,12}; x = Matrices.solve(A,b); Matrices.eigenValues(A);</pre>	<p>Modelica.Math.Matrices / Modelica.Utilities</p> <p>Functions operating on matrices, e.g. to solve linear systems and compute eigen and singular values. Also functions are provided to operate on strings, streams and files.</p>
  <pre>type Angle = Real (final quantity = "Angle", final unit = "rad", displayUnit = "deg");</pre>	<p>Modelica.Constants, Modelica.Icons, Modelica.SIunits</p> <p>Utility libraries to provide</p> <ul style="list-style-type: none"> • often used constants such as e, π, R • a library of icons that can be used in models • about 450 predefined types, such as Mass, Angle, Time, based on the international standard on units

The Modelica standard library defines the most important *elementary connectors* in various domains. If any possible, a user should utilize these connectors in order that components from the Modelica Standard Library and from other libraries can be combined without problems. The following elementary connectors are defined (potential variables are connector variables without the flow attribute, flow variables are connector variables that have the flow attribute):

Domain	Potential variables	Flow Variables	Connector Definition	Icons
electrical analog	electrical potential	electrical current	Modelica.Electrical.Analog.Interfaces .Pin, .PositivePin, .NegativePin	 
electrical multi-phase	vector of electrical pins		Modelica.Electrical.MultiPhase.Interfaces .Plug, .PositivePlug, .NegativePlug	 
electrical sphase phasor	2 electrical potentials	2 electrical currents	Modelica.Electrical.Machines.Interfaces SpacePhasor	
electrical digital	Integer (1..9)	---	Modelica.Electrical.Digital.Interfaces .DigitalSignal, .DigitalInput, .DigitalOutput	 
translational	distance	cut-force	Modelica.Mechanics.Translational.Interfaces .Flange_a, .Flange_b	 
rotational	angle	cut-torque	Modelica.Mechanics.Rotational.Interfaces .Flange_a, .Flange_b	 
3-dim. mechanics	position vector orientation object	cut-force vector cut-torque vector	Modelica.Mechanics.MultiBody.Interfaces .Frame, .Frame_a, .Frame_b, .Frame_resolve	 
simple fluid flow	pressure specific enthalpy	mass flow rate enthalpy flow rate	Modelica.Thermal.FluidHeatFlow.Interfaces .FlowPort, .FlowPort_a, .FlowPort_b	 
heat transfer	temperature	heat flow rate	Modelica.Thermal.HeatTransfer.Interfaces .HeatPort, .HeatPort_a, .HeatPort_b	 
block diagram	Real, Integer, Boolean		Modelica.Blocks.Interfaces .RealSignal, .RealInput, .RealOutput .IntegerSignal, .IntegerInput, .IntegerOutput .BooleanSignal, .BooleanInput, .BooleanOutput	     
state machine	Boolean variables (occupied, set, available, reset)		Modelica.StateGraph.Interfaces .Step_in, .Step_out, .Transition_in, .Transition_out	 
Connectors from libraries that will be included in the future in to package Modelica				
thermo fluid flow	pressure spec.enthalpy mass fractions	mass flow rate enthalpy flow rate subst. mass flow rates	Modelica_Fluid.Interfaces .FluidPort, .FluidPort_a, .FluidPort_b	 
magnetic	magnetic potential	magnetic flux	Magnetic.Interfaces .MagneticPort, .PositiveMagneticPort, .NegativeMagneticPort	 

In all domains, usually 2 connectors are defined. The variable declarations are *identical*, only the icons are different in order that it is easy to distinguish connectors of the same domain that are attached at the same component.

Appendix A

Glossary

- algorithm section:** part of a class definition consisting of the keyword `algorithm` followed by a sequence of statements. Like an equation, an algorithm section relates variables, i.e. constrains the values that these variables can take simultaneously. In contrast to an equation section, an algorithm section distinguishes inputs from outputs: An algorithm section specifies how to compute output variables as a function of given input variables. A Modelica processor may actually invert an algorithm section, i.e. compute inputs from given outputs, e.g. by search (generate and test), or by deriving an inverse algorithm symbolically. (See Chapter 11.)
- array** or **array variable:** a component whose components are array elements. For an array, the ordering of its components matters: The k th element in the sequence of components of an array x is the array element with index k , denoted $x[k]$. All elements of an array have the same type. An array element may again be an array, i.e. arrays can be nested. An array element is hence referenced using n indices in general, where n is the number of dimensions of the array. Special cases are matrix ($n=2$) and vector ($n=1$). Integer indices start with 1, not zero. (See Chapter 10.)
- array constructor:** an array can be built using the array-function – with the shorthand $\{a, b, \dots\}$, and can also include an iterator to build an array of expressions. (See Section 10.4.)
- array element:** a component contained in an array. An array element has no identifier. Instead they are referenced by array access expressions called indices that use enumeration values or positive integer index values. (See Chapter 10.)
- assignment:** a statement of the form $x := \text{expr}$. The expression expr must not have higher variability than x . (See Section 11.2.1.)
- attribute:** a component contained in a scalar component, such as `min`, `max`, and `unit`. All attributes are predefined and attribute values can only be defined using a modification, such as in `Real x(unit="kg")`. Attributes cannot be accessed using dot notation, and are not constrained by equations and algorithm sections. E.g. in `Real x(unit="kg") = y;` only the values of x and y are declared to be equal, but not their unit attributes, nor any other attribute of x and y . (See Section 4.8.)
- base class:** class A is called a base class of B , if class B extends class A . This relation is specified by an `extends` clause in B or in one of B 's base classes. A class inherits all elements from its base classes, and may modify all non-final elements inherited from base classes. (See Section 7.1.)
- binding equation:** Either a declaration equation or an element modification for the value of the variable. A component with a binding equation has its value bound to some expression. (See Section 8.1.)
- class:** a description that generates an object called instance. The description consists of a class definition, an modification environment that modifies the class definition, an optional list of dimension expressions if the class is an array class, and a lexically enclosing class for all classes. (See Section 4.5.)
- class type** or **inheritance interface:** property of a class, consisting of a number of attributes and a set of public or protected elements consisting of element name, element type, and element attributes. (See Section 6.2.2.)
- component** or **variable:** an instance (object) generated by a component declaration. Special kinds of components are scalar, array, and attribute. (See Section 4.4.)
- component declaration:** an element of a class definition that generates a component. A component declaration specifies (1) a component name, i.e., an identifier, (2) the class to be flattened in order to generate the component, and (3) an optional Boolean parameter expression. Generation of the component is suppressed if

this parameter expression evaluates to false. A component declaration may be overridden by an element redeclaration. (See Section 4.4.)

component reference: An expression containing a sequence of identifiers and indices. A component reference is equivalent to the referenced object, which must be a component. A component reference is resolved (evaluated) in the scope of a class (or expression for the case of a local iterator variable). A scope defines a set of visible components and classes. Example reference: `Ele.Resistor.u[21].r` (See Sections 4.4 and 10.6.9.)

declaration assignment: assignment of the form `x := expression` defined by a component declaration. This is similar to a declaration equation. In contrast to a declaration equation, a declaration assignment is allowed only when declaring a component contained in a function. (See Section 12.4.3.)

declaration equation: Equation of the form `x = expression` defined by a component declaration. The expression must not have higher variability than the declared component `x`. Unlike other equations, a declaration equation can be overridden (replaced or removed) by an element modification. (See Section 4.4.2.1.)

derived class or subclass or extended class: class `B` is called derived from `A`, if `B` extends `A`. (See Chapter 7.)

element: part of a class definition, one of: class definition, component declaration, or extends clause. Component declarations and class definitions are called named elements. An element is either inherited from a base class or local.

element modification: part of a modification, overrides the declaration equation in the class used by the instance generated by the modified element. Example: `vcc(unit="V")=1000`. (See Section 7.2.)

element redeclaration: part of a modification, replaces one of the named elements possibly used to build the instance generated by the element that contains the redeclaration. Example: `redeclare type Voltage = Real(unit="V")` replaces `type Voltage`. (See Section 7.3.)

equation: part of a class definition. A scalar equation relates scalar variables, i.e. constrains the values that these variables can take simultaneously. When $n-1$ variables of an equation containing n variables are known, the value of the n th variable can be inferred (solved for). In contrast to a statement in an algorithm section, an equation does not define for which of its variable it is to be solved. Special cases are: initial equations, instantaneous equations, declaration equations. (See Chapter 8.)

event: something that occurs instantaneously at a specific time or when a specific condition occurs. Events are for example defined by the condition occurring in a when clause, if clause, or if expression. (See Section 8.5.)

expression: a term built from operators, function references, components, or component references (referring to components) and literals. Each expression has a type and a variability. (See Chapter 3.)

extends clause: an unnamed element of a class definition that uses a name and an optional modification to specify a base class of the class defined using the class definition. (See Chapter 7.)

flattening: the computation that creates a flattened class of a given class, where all inheritance, modification, etc. has been performed and all names resolved, consisting of a flat set of equations, algorithm sections, component declarations, and functions. (See Section 5.6.)

function: a class of the specialized class function. (See Chapter 12.)

function subtype or function compatible interface: `A` is a function subtype of `B` iff `A` is a subtype of `B` and the additional arguments of function `A` that are not in function `B` are defined in such a way (e.g. additional arguments need to have default values), that `A` can be called at places where `B` is called. (See Section 6.5.)

identifier or ident: an atomic (not composed) name. Example: `Resistor` (See Section 2.3.)

index or subscript: An expression, typically of Integer type or the colon symbol (`:`), used to reference a component (or a range of components) of an array. (See Section 10.5.)

inheritance interface or **class type**: property of a class, consisting of a number of attributes and a set of public or protected elements consisting of element name, element type, and element attributes. (See Section 6.2.2.)

instance: the object generated by a class. An instance contains zero or more components. An instance may also be associated with equations and algorithms. An instance has a type. Basically, two instances have same type, if their important attributes are the same and their public components and classes have pair wise equal identifiers and types. More specific type equivalence definitions are given e.g. for functions.

instantaneous: An equation or statement is instantaneous if it holds only at events, i.e., at single points in time. The equations and statements of a when-clause are instantaneous. (See Sections 8.3.5 and 11.2.7.)

interface: see type. (See Section 6.2.)

literal: a real, integer, boolean, enumeration, or string literal. Used to build expressions. (See Section 2.4.)

matrix: an array where the number of dimensions is 2. (See Chapter 10.)

modification: part of an element. Modifies the instance generated by that element. A modification contains element modifications and element redeclarations. (See Section 7.2.)

modification environment: the modification environment of a class defines how to modify the corresponding class definition when flattening the class. (See Section 7.2.2.)

name: Sequence of one or more identifiers. Used to reference a class. A class name is resolved in the scope of a class, which defines a set of visible classes. Example name: "Ele.Resistor". (See Section 2.3.2.)

partial flattening: first find the names of declared local classes and components. Modifiers, if present, are merged to the local elements and redeclarations are performed. Then base-classes are looked up, flattened and inserted into the class. See also flattening, which additionally flattens local elements and performs modifications. (See Section 5.6.1.)

plug-compatibility: see restricted subtyping and Section 6.4.

predefined type: one of the types Real, Boolean, Integer, String and types defined as enumeration types. The component declarations of the predefined types define attributes such as min, max, and unit. (See Section 4.8.)

prefix: property of an element of a class definition which can be present or not be present, e.g. final, public, flow. (See Section 4.4.2.2.)

primitive type: one of the built-in types RealType, BooleanType, IntegerType, StringType, EnumType. The primitive types are used to define attributes and value of predefined types and enumeration types. (See Section 4.8.)

restricted subtyping or **plug-compatibility**: a type A is a restricted subtype of type B iff A is a subtype of B, and all public components present in A but not in B must be default-connectable. This is used to avoid introducing, via a redeclaration, an un-connected connector in the object/class of type A at a level where a connection is not possible. (See Section 6.4.)

scalar or scalar variable: a variable that is not an array.

specialized class: one of: model, connector, package, record, block, function, type. The class restriction of a class represents an assertion regarding the content of the class and restricts its use in other classes. For example, a class having the package class restriction must only contain classes and constants. (See Section 4.6.)

subtype or **interface compatible**: relation between types. A is a subtype of (interface compatible with) B iff a number of properties of A and B are the same and all important elements of B have corresponding elements in A with the same names and their types being subtypes of the corresponding element types in B. See also restricted subtyping and function restricted subtyping. (See Section 6.3.)

supertype: relation between types. The inverse of subtype. A is a subtype of B means that B is a supertype of A. (See Section 6.3.)

transitively nonreplaceable: a class reference is considered transitively non-replaceable if there are no replaceable elements in the referenced class, or any of its base classes or constraining types transitively at any level. (See Section 6.2.1.)

type or **interface:** property of an instance, expression, consisting of a number of attributes and a set of public elements consisting of element name, element type, and element attributes. Note: The concept of class type is a property of a class definition. (See Section 6.2.)

variability: property of an expression: one of

- **continuous:** an expression that may change its value at any point in time.
- **discrete:** may change its value only at events during simulation.
- **parameter:** constant during the entire simulation, recommended to change for a component.
- **constant:** constant during the entire simulation (can be used in a package) .

Assignments $x := \text{expr}$ and binding equations $x = \text{expr}$ must satisfy a variability constraint: The expression must not have a higher variability than component x . (See Section 3.8.)

variable: synonym for component. (See Section 4.4.)

vector: an array where the number of dimensions is 1. (See Chapter 10.)

Appendix B

Modelica Concrete Syntax

B.1 Lexical conventions

The following syntactic meta symbols are used (extended BNF):

```
[ ] optional
{ } repeat zero or more times
| or
```

The following lexical units are defined:

```
IDENT = NONDIGIT { DIGIT | NONDIGIT } | Q-IDENT
Q-IDENT = "'" ( Q-CHAR | S-ESCAPE ) { Q-CHAR | S-ESCAPE } "'"
NONDIGIT = "_" | letters "a" to "z" | letters "A" to "Z"
STRING = "\"" { S-CHAR | S-ESCAPE } "\""
S-CHAR = any member of the source character set except double-quote "\"", and backslash "\"
Q-CHAR = any member of the source character set except single-quote "'", and backslash "\"
S-ESCAPE = "\"'\" | "\"\" | "\"?\" | "\"\\\" |
           "\"a\" | "\"b\" | "\"f\" | "\"n\" | "\"r\" | "\"t\" | "\"v\"
DIGIT = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
UNSIGNED_INTEGER = DIGIT { DIGIT }
UNSIGNED_NUMBER = UNSIGNED_INTEGER [ "." [ UNSIGNED_INTEGER ] ]
                  [ ( "e" | "E" ) [ "+" | "-" ] UNSIGNED_INTEGER ]
```

[The single quotes are part of an identifier. E.g. 'x' and x are different IDENTs].

Note: string constant concatenation "a" "b" becoming "ab" (as in C) is replaced by the "+" operator in Modelica.

Modelica uses the same comment syntax as C++ and Java, and also has structured comments in the form of annotations and string comments. Inside a comment, the sequence <HTML> </HTML> indicates HTML code which may be used by tools to facilitate model documentation.

Boldface denotes keywords of the Modelica language. Keywords are reserved words and may not be used as identifiers, with the exception of *initial* which is a keyword in section headings, but it is also possible to call the function *initial()*.

B.2 Grammar

B.2.1 Stored Definition – Within

```
stored_definition:
  [ within [ name ] ";" ]
  { [ final ] class_definition ";" }
```

B.2.2 Class Definition

```
class_definition :
  [ encapsulated ]
  [ partial ]
  ( class | model | record | block | [ expandable ] connector | type |
```

```
    package | function )
    class_specifier

class_specifier :
    IDENT string_comment composition end IDENT
  | IDENT "=" base_prefix name [ array_subscripts ]
    [ class_modification ] comment
  | IDENT "=" enumeration "(" ( [enum_list] | ":" ) ")" comment
  | IDENT "=" der "(" name "," IDENT { "," IDENT } ")" comment
  | extends IDENT [ class_modification ] string_comment composition
    end IDENT

base_prefix :
    type_prefix

enum_list    : enumeration_literal { "," enumeration_literal }

enumeration_literal : IDENT comment

composition :
    element_list
  { public element_list |
    protected element_list |
    equation_section |
    algorithm_section
  }
  [ external [ language_specification ]
    [ external_function_call ] [ annotation ";" ]
    [ annotation ";" ] ]

language_specification :
    STRING

external_function_call :
    [ component_reference "=" ]
    IDENT "(" [ expression_list ] ")"

element_list :
    { element ";" | annotation ";" }

element :
    import_clause |
    extends_clause |
    [ redeclare ]
    [ final ]
    [ inner ] [ outer ]
    ( ( class_definition | component_clause ) |
      replaceable ( class_definition | component_clause )
      [constraining_clause comment])

import_clause :
    import ( IDENT "=" name | name [ "." "*" ] ) comment
```

B.2.3 Extends

```
extends_clause :
    extends name [ class_modification ] [annotation]

constraining_clause :
    constrainedby name [ class_modification ]
```

B.2.4 Component Clause

```
component_clause:
    type_prefix type_specifier [ array_subscripts ] component_list
```

```

type_prefix :
  [ flow ]
  [ discrete | parameter | constant ] [ input | output ]

type_specifier :
  name

component_list :
  component_declaration { "," component_declaration }

component_declaration :
  declaration [ conditional_attribute ] comment

conditional_attribute:
  if expression

declaration :
  IDENT [ array_subscripts ] [ modification ]

```

B.2.5 Modification

```

modification :
  class_modification [ "=" expression ]
  | "=" expression
  | "!=" expression

class_modification :
  "(" [ argument_list ] ")"

argument_list :
  argument { "," argument }

argument :
  element_modification_or_replaceable
  | element_redeclaration

element_modification_or_replaceable:
  [ each ] [ final ] ( element_modification | element_replaceable )

element_modification :
  component_reference [ modification ] string_comment

element_redeclaration :
  redeclare [ each ] [ final ]
  ( ( class_definition | component_clause1 ) | element_replaceable )

element_replaceable:
  replaceable ( class_definition | component_clause1 )
  [constraining_clause]

component_clause1 :
  type_prefix type_specifier component_declaration1

component_declaration1 :
  declaration comment

```

B.2.6 Equations

```

equation_section :
  [ initial ] equation { equation ";" | annotation ";" }

algorithm_section :
  [ initial ] algorithm { statement ";" | annotation ";" }

```

```
equation :
  ( simple_expression "=" expression
  | if_equation
  | for_equation
  | connect_clause
  | when_equation
  | IDENT function_call_args )
  comment

statement :
  ( component_reference ( "!=" expression | function_call_args )
  | "(" output_expression_list ")" "!=" component_reference function_call_args
  | break
  | return
  | if_statement
  | for_statement
  | while_statement
  | when_statement )
  comment

if_equation :
  if expression then
    { equation ";" }
  { elseif expression then
    { equation ";" }
  }
  [ else
    { equation ";" }
  ]
  end if

if_statement :
  if expression then
    { statement ";" }
  { elseif expression then
    { statement ";" }
  }
  [ else
    { statement ";" }
  ]
  end if

for_equation :
  for for_indices loop
    { equation ";" }
  end for

for_statement :
  for for_indices loop
    { statement ";" }
  end for

for_indices :
  for_index { "," for_index }

for_index:
  IDENT [ in expression ]

while_statement :
  while expression loop
    { statement ";" }
  end while

when_equation :
  when expression then
    { equation ";" }
  { elsewhen expression then
    { equation ";" } }
```

```

    end when

when_statement :
    when expression then
        { statement ";" }
    { elseif expression then
        { statement ";" } }
    end when

connect_clause :
    connect "(" component_reference "," component_reference ")"

```

B.2.7 Expressions

```

expression :
    simple_expression
    | if expression then expression { elseif expression then expression }
      else expression

simple_expression :
    logical_expression [ ":" logical_expression [ ":" logical_expression ] ]

logical_expression :
    logical_term { or logical_term }

logical_term :
    logical_factor { and logical_factor }

logical_factor :
    [ not ] relation

relation :
    arithmetic_expression [ rel_op arithmetic_expression ]

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

arithmetic_expression :
    [ add_op ] term { add_op term }

add_op :
    "+" | "-" | ".*" | ".-"

term :
    factor { mul_op factor }

mul_op :
    "*" | "/" | ".*" | "./"

factor :
    primary [ ("^" | ".^") primary ]

primary :
    UNSIGNED_NUMBER
    | STRING
    | false
    | true
    | name function_call_args
    | component_reference
    | "(" output_expression_list ")"
    | "[" expression_list { ";" expression_list } "]"
    | "{" function_arguments "}"
    | end

name :
    IDENT [ "." name ]

```

```
component_reference :  
  IDENT [ array_subscripts ] [ "." component_reference ]  
  
function_call_args :  
  "(" [ function_arguments ] ")"  
  
function_arguments :  
  expression [ "," function_arguments | for for_indices ]  
  | named_arguments  
  
named_arguments: named_argument [ "," named_arguments ]  
  
named_argument: IDENT "=" expression  
  
output_expression_list:  
  [ expression ] { "," [ expression ] }  
  
expression_list :  
  expression { "," expression }  
  
array_subscripts :  
  "[" subscript { "," subscript } "]"  
  
subscript :  
  ":" | expression  
  
comment :  
  string_comment [ annotation ]  
  
string_comment :  
  [ STRING { "+" STRING } ]  
  
annotation :  
  annotation class_modification
```


Appendix C

Modelica DAE Representation

In this appendix, the mapping of a Modelica model into an appropriate mathematical description form is discussed.

In a first step, a Modelica translator transforms a hierarchical Modelica simulation model into a “flat” set of Modelica “statements”, consisting of the equation and algorithm sections of all used components by:

- expanding all class definitions (flattening the inheritance tree) and adding the equations and assignment statements of the expanded classes for every instance of the model.
- replacing all connect-equations by the corresponding equations of the connection set (see Section 9.2).
- mapping all algorithm sections to equation sets.
- mapping all when-clauses to equation sets (see Section 8.3.5).

As a result of this transformation process, a set of equations is obtained consisting of differential, algebraic and discrete equations of the following form where ($\mathbf{v} := [\dot{\mathbf{x}}; \mathbf{x}; \mathbf{y}; t; \mathbf{m}; \text{pre}(\mathbf{m}); \mathbf{p}]$):

$$(1a) \quad \mathbf{c} := \mathbf{f}_c(\text{relation}(\mathbf{v}))$$

$$(1b) \quad \mathbf{m} := \mathbf{f}_m(\mathbf{v}, \mathbf{c})$$

$$(1c) \quad \mathbf{0} = \mathbf{f}_x(\mathbf{v}, \mathbf{c})$$

and where

\mathbf{p}	Modelica variables declared as <i>parameter</i> or <i>constant</i> , i.e., variables without any time-dependency.
t	Modelica variable <i>time</i> , the independent (real) variable.
$\mathbf{x}(t)$	Modelica variables of type <i>Real</i> , appearing differentiated.
$\mathbf{m}(t_e)$	Modelica variables of type <i>discrete Real</i> , <i>Boolean</i> , <i>Integer</i> which are unknown. These variables change their value only at event instants t_e . $\text{pre}(\mathbf{m})$ are the values of \mathbf{m} immediately before the current event occurred.
$\mathbf{y}(t)$	Modelica variables of type <i>Real</i> which do not fall into any other category (= algebraic variables).
$\mathbf{c}(t_e)$	The conditions of all if-expressions generated including when-clauses after conversion, see Section 8.3.5).
$\text{relation}(\mathbf{v})$	A relation containing variables v_i , e.g. $v_1 > v_2$, $v_3 \geq 0$.

For simplicity, the special cases of the `noEvent()` operator and of the `reinit()` operator are not contained in the equations above and are not discussed below.

The generated set of equations is used for simulation and other analysis activities. Simulation means that an initial value problem is solved, i.e., initial values have to be provided for the states \mathbf{x} , Section 8.6. The equations define a DAE (Differential Algebraic Equations) which may have discontinuities, a variable structure and/or which are controlled by a discrete-event system. Such types of systems are called *hybrid DAEs*. Simulation is performed in the following way:

1. The DAE (1c) is solved by a numerical integration method. In this phase the conditions \mathbf{c} of the if- and when-clauses, as well as the discrete variables \mathbf{m} are kept constant. Therefore, (1c) is a continuous function of continuous variables and the most basic requirement of numerical integrators is fulfilled.
2. During integration, all relations from (1a) are monitored. If one of the relations changes its value an event is triggered, i.e., the exact time instant of the change is determined and the integration is halted. As

discussed in Section 8.5, relations which depend only on time are usually treated in a special way, because this allows to determine the time instant of the next event in advance.

3. At an event instant, (1) is a mixed set of algebraic equations which is solved for the Real, Boolean and Integer unknowns.
4. After an event is processed, the integration is restarted with 1.

Note, that both the values of the conditions c as well as the values of m (all discrete Real, Boolean and Integer variables) are only changed at an event instant and that these variables remain constant during continuous integration. At every event instant, new values of the discrete variables m and of new initial values for the states x are determined. The change of discrete variables may characterize a new structure of a DAE where elements of the state vector x are *disabled*. In other words, the number of state variables, algebraic variables and residue equations of a DAE may change at event instants by disabling the appropriate part of the DAE. For clarity of the equations, this is not explicitly shown by an additional index in (1).

At an event instant, including the initial event, the model equations are reinitialized according to the following iteration procedure:

```
known variables:  $\mathbf{x}$ ,  $t$ ,  $\mathbf{p}$ 
unknown variables:  $d\mathbf{x}/dt$ ,  $\mathbf{y}$ ,  $\mathbf{m}$ ,  $\text{pre}(\mathbf{m})$ ,  $c$ 

//  $\text{pre}(\mathbf{m})$  = value of  $\mathbf{m}$  before event occurred
loop
  solve (1) for the unknowns, with  $\text{pre}(\mathbf{m})$  fixed
  if  $\mathbf{m} == \text{pre}(\mathbf{m})$  then break
   $\text{pre}(\mathbf{m}) := \mathbf{m}$ 
end loop
```

Solving (1) for the unknowns is non-trivial, because this set of equations contains not only Real, but also Boolean and Integer unknowns. Usually, in a first step these equations are sorted and in many cases the Boolean and Integer unknowns can be just computed by a forward evaluation sequence. In some cases, there remain systems of equations (e.g. for ideal diodes, Coulomb friction elements) and specialized algorithms have to be used to solve them.

Due to the construction of the equations by "flattening" a Modelica model, the hybrid DAE (1) contains a huge number of sparse equations. Therefore, direct simulation of (1) requires sparse matrix methods. However, solving this initial set of equations directly with a numerical method is both unreliable and inefficient. One reason is that many Modelica models, like the mechanical ones, have a DAE index of 2 or 3, i.e., the overall number of states of the model is less than the sum of the states of the sub-components. In such a case, every direct numerical method has the difficulty that the numerical condition becomes worse, if the integrator step size is reduced and that a step size of zero leads to a singularity. Another problem is the handling of idealized elements, such as ideal diodes or Coulomb friction. These elements lead to mixed systems of equations having both Real and Boolean unknowns. Specialized algorithms are needed to solve such systems.

To summarize, symbolic transformation techniques are needed to transform (1) in a set of equations which can be numerically solved reliably. Most important, the algorithm of Pantelides should to be applied to differentiate certain parts of the equations in order to reduce the index. Note, that also explicit integration methods, such as Runge-Kutta algorithms, can be used to solve (1c), after the index of (1c) has been reduced by the Pantelides algorithm: During continuous integration, the integrator provides x and t . Then, (1c) is a linear or nonlinear system of equations to compute the algebraic variables y and the state derivatives dx/dt and the model returns dx/dt to the integrator by solving these systems of equations. Often, (1c) is just a linear system of equations in these unknowns, so that the solution is straightforward. This procedure is especially useful for real-time simulation where usually explicit one-step methods are used.

Appendix D

Modelica Revision History

This appendix describes the history of the Modelica Language Design, and its contributors. The current version of this document is available from <http://www.modelica.org/>.

D.1 Modelica 3.0

Modelica 3.0 was released Sept. 5, 2007. The Modelica 3.0 specification was edited by Peter Fritzson, Hans Olsson, and Martin Otter.

D.1.1 Contributors to the Modelica Language, Version 3.0

The Modelica 3.0 specification was newly structured and written by Peter Fritzson using text from the previous specification and also adding new explanatory text. This draft specification was afterwards improved by members of the Modelica Association.

The concept of “balanced models” (which is the major change of the language) was developed by Hans Olsson with contributions from Martin Otter, Hilding Elmqvist, and Sven Erik Mattsson.

The graphical annotations have been redesigned and improved by Daniel Hedberg, Erik Areskog, Dag Brück, and Hilding Elmqvist with contributions from Peter Aronsson and Gerd Kurzbach.

The following members of the Modelica Association participated at design meetings and contributed to the Modelica 3.0 specification:

Peter Aronsson, MathCore AB, Linköping, Sweden
Bernhard Bachmann, University of Applied Sciences, Bielefeld, Germany
John Batteh, Ford Motor Company, Dearborn, MI, U.S.A.
David Broman, Linköping University, Linköping, Sweden
Peter Bunus, Linköping University, Linköping, Sweden
Dag Brück, Dynasim, Lund, Sweden
Francesco Casella, Politecnico di Milano, Milano, Italy
Christoph Clauß, Fraunhofer Institute for Integrated Circuits, Dresden, Germany
Thomas Doumenc, Dassault Systèmes, Paris, France
Jonas Eborn, Modelon AB, Lund, Sweden
Hilding Elmqvist, Dynasim, Lund, Sweden
Rüdiger Franke, ABB Corporate Research, Ladenburg, Germany
Peter Fritzson, Linköping University, Sweden
Sebastien Furic, Imagine, Roanne, France
Anton Haumer, Technical Consulting & Electrical Engineering, St.Andrae-Woerden, Austria
Daniel Hedberg, MathCore AB, Linköping, Sweden
Carsten Heinrich, Institut für Luft- und Kältetechnik gGmbH, Dresden, Germany
Olof Johansson, Linköping University, Linköping, Sweden
Christian Kral, arsenal research, Vienna, Austria
Roland Kossel, TLK Thermo GmbH, Braunschweig, Germany
Gerd Kurzbach, ITI GmbH, Dresden, Germany
Christian Kral, arsenal research, Vienna, Austria
Kilian Link, Siemens AB, Erlangen, Germany
José Diaz Lopez, Dynasim AB, Lund, Sweden

Karin Lund, Fachhochschule Ulm, Germany
 Håkan Lundvall, Linköping University, Linköping, Sweden
 Ludwig Marvan, VA TECH ELIN EBG Elektronik GmbH & Co, Vienna, Austria
 Sven Erik Mattsson, Dynasim, Lund, Sweden
 Jakob Mauss, Qtronic GmbH, Berlin, Germany
 Chuck Newman, Ford Motor Company, Dearborn, MI, U.S.A.
 Kaj Nyström, Linköping University, Linköping, Sweden
 Hans Olsson, Dynasim, Lund, Sweden
 Martin Otter, German Aerospace Center, Oberpfaffenhofen, Germany
 Markus Plainer, Arsenal Research, Vienna, Austria
 Adrian Pop, Linköping University, Linköping, Sweden
 Katrin Pröhl, Technical University Hamburg-Harburg, Germany
 Christoph Richter, Technical University of Braunschweig, Braunschweig, Germany
 Anders Sandholm, Linköping University, Linköping, Sweden
 Christian Schweiger, German Aerospace Center, Oberpfaffenhofen, Germany
 Michael Tiller, Ford Motor Company/Emmeskay, Dearborn, MI, U.S.A
 Hubertus Tummescheit, Modelon AB, Lund, Sweden
 Hans-Jürg Wiesmann, ABB Switzerland Ltd., Corporate Research, Baden, Switzerland

D.1.2 Main Changes in Modelica 3.0

Modelica 3.0 is a “clean-up” version of the Modelica language. For example, the specification is newly written to define the language in a better way, errors in the language are fixed, unclear or undefined items are more precisely described, and mild restrictions are introduced into the language in order that a Modelica translator can much earlier and more precisely detect modeling errors. Automated conversion of models to Modelica 3.0 is possible. Furthermore, a tool can potentially handle much larger models.

This Modelica version is for the first time (slightly) not backward compatible to previous versions (all previous versions have been backward compatible with exception of tiny issues as newly introduced keywords). As a result, e.g., “unsafe” models of previous Modelica versions are no longer valid. It is expected that Modelica tool vendors provide (semi-) automatic mechanisms for conversion of models and libraries.

The following main changes in Modelica 3.0 are **not backwards compatible**:

- Restrictions to connectors (see Section 9.3): For each non-partial connector class the number of flow variables shall be equal to the number of variables that are neither parameter, constant, input, output, nor flow. For example, the following connector is illegal in Modelica 3 :

```

connector notValid // illegal connector
  Real r1;
  Real r2;
  flow Real r3;
end notValid;

```
- In a non-partial model or block, all non-connector inputs of model or block components must have binding equations.
- A component declared with the inner or outer prefix shall not be of a class having top-level public connectors containing inputs.
- Modifiers for components shall only contain redeclarations of replaceable elements and binding equations for parameters, constants, inputs and variables having a default binding equation.
- All non-partial model and block classes must be locally balanced (see Section 4.7). This means that the local number of unknowns equals the local equation size. Together with other restrictions, this leads to the strong property that a simulation model is always globally balanced (i.e., the number of unknowns is equal to the number of equations).
- Prefixes input, output, inner, outer, flow are not allowed in a record

- The built-in operators “abs(...)” and “sign(...)” do no longer generate events but are implicitly defined with a `noEvent(...)` operator.
- The constraining clause of a replaceable class or component is changed from keyword `extends` to the new keyword `constrainedby` (since the `extends` keyword could lead to the wrong impression that the redeclared model must inherit from the constraining class, but this is not the case).

The following changes in the Modelica 3.0 graphical annotations are also **not backwards compatible**:

- Changed the definition of icon placement (record `Transformation`), so that the actual coordinates of the icon of a model instance are defined in the class where the instance is defined and no longer in the class where the icon is defined (this was a flaw in the Modelica 2.0 graphical annotations).
- Improved the definition of the rotation of a graphical primitive.
- Change `fontSize` unit from `DrawingUnit` to `pt` (since this is the usual unit for fonts).

The following main changes in Modelica 3.0 are **backwards compatible**:

- New element-wise operators: `., .-, .*, ./, .^`.
- A third argument `AssertionLevel` to built-in function `assert(...)` in order that warnings can optionally be defined.
- New annotations section 14.1 “Vendor-Specific Annotations”:
In this section it is precisely defined how vendor-specific annotations should be marked. Any tool shall save files with all standard annotations (defined in Chapter 14) and all vendor-specific annotations intact. The advantage is that a typo in non-vendor annotations can now be detected and marked as an error, whereas in previous versions this had to be ignored.
- New annotation in section 14.2 “Annotations for Documentation”:
`preferredView = info, diagram or text`
- New annotations section **Fehler! Verweisquelle konnte nicht gefunden werden.** “Annotations for Code Generation”:
`Evaluate, HideResult, Inline, LateInline, smoothOrder`
- New annotation section 14.4 “Annotations for Simulation Experiments”:
`StartTime, StopTime, Tolerance` to define important parameters of an experiment setup.
- New annotations for graphical annotations in section 14.5:
New attribute `Smooth = enumeration(None, Bezier)` for graphical objects and connection lines (Bezier defines a Bezier spline).
New attribute `visible` in record `Placement` allows to make a graphical annotation invisible (e.g. after inheritance).
New attributes `startAngle, endAngle` in record `ellipse` to define part of an ellipse.
New layer specific annotations `IconMap` and `DiagramMap` for extends.
New attribute `horizontalAlignment` to the `Text` record to define the horizontal alignment of text.
- New annotations for schematic animation and interactive user input in section 14.5:
`DynamicSelect(..)` to modify annotation literals by the actual values of variables.
`OnMouseDownSetBoolean, OnMouseUpSetBoolean, OnMouseMoveXSetReal, OnMouseMoveYSetReal, OnMouseDownEditReal, OnMouseDownEditString` to interactively set the variable of a class during simulation.

The following **errors** have been **fixed** in the Modelica specification:

- Syntax rule for a function call (e.g., according to the grammar in Modelica 2.2 a function call of the form `Modelica.Math.Matrices.eig(...)` was invalid because the function name could not have “.”. However, all Modelica tools supported the desired “full Modelica name” also for function calls).

D.2 Modelica 2.2

Modelica 2.2 was released February 2, 2005. The Modelica 2.2 specification was edited by Hans Olsson, Michael Tiller and Martin Otter.

D.2.1 Contributors to the Modelica Language, Version 2.2

Bernhard Bachmann , University of Applied Sciences, Bielefeld, Germany
 John Batteh, Ford Motor Company, Dearborn, MI, U.S.A.
 Dag Brück, Dynasim, Lund, Sweden
 Francesco Casella, Politecnico di Milano, Milano, Italy
 Christoph Clauß, Fraunhofer Institute for Integrated Circuits, Dresden, Germany
 Jonas Eborn, Modelon AB, Lund, Sweden
 Hilding Elmqvist, Dynasim, Lund, Sweden
 Rüdiger Franke, ABB Corporate Research, Ladenburg, Germany
 Peter Fritzson, Linköping University, Sweden
 Anton Haumer, Technical Consulting & Electrical Engineering, St.Andrae-Woerden, Austria
 Christian Kral, arsenal research, Vienna, Austria
 Sven Erik Mattsson, Dynasim, Lund, Sweden
 Chuck Newman, Ford Motor Company, Dearborn, MI, U.S.A.
 Hans Olsson, Dynasim, Lund, Sweden
 Martin Otter, German Aerospace Center, Oberpfaffenhofen, Germany
 Markus Plainier, Arsenal Research, Vienna, Austria
 Adrian Pop, Linköping University, Sweden
 Katrin Pröhl, Technical University Hamburg-Harburg, Germany
 André Schneider, Fraunhofer Institute for Integrated Circuits, Dresden, Germany
 Christian Schweiger, German Aerospace Center, Oberpfaffenhofen, Germany
 Michael Tiller, Ford Motor Company, Dearborn, MI, U.S.A.
 Hubertus Tummescheit, Modelon AB, Lund, Sweden
 Hans-Jürg Wiesmann, ABB Switzerland Ltd.,Corporate Research, Baden, Switzerland

D.2.2 Main Changes in Modelica 2.2

The main changes in Modelica 2.2 are:

- Conditional component declarations to ignore component declarations depending on a parameter expression. Connection equations that reference a component that is no longer present, are ignored.
- In redeclarations some parts of the original declaration are automatically inherited by the new declaration. This is intended to make it easier to write declarations by not having to repeat common parts of the declarations, and does in particular apply to attributes that must be identical.
- Recursive `inner/outer` definitions to define hierarchically structured inner/outer declarations that can communicate with each other: An element declared with both the prefixes `inner` and `outer` conceptually introduces two declarations with the same name, one that follows the rules for `inner` and another that follows the rules for `outer`.
- A non-input array component declared in a function with a dimension size specified by `colon(:)` and no declaration assignment, can change size in the function in a simple and convenient way.
- A new type of connector, called “expandable connector” was introduced. This connector has less strict requirements about name matching of connected connectors and can be used conveniently in situations that required replaceable connectors previously. One main application area is to construct signal buses of complex systems.
- The derivative operator `der(expr)` may have an expression as argument and not only a variable name as previously, e.g., `der(m*h)` is interpreted as `der(m)*h + m*der(h)`.

- A function can be defined as partial derivative of another function, e.g.:
- `"function Gibbs_T = der(Gibbs,T)"` is a function that computes the partial derivative of function Gibbs with respect to its input argument T.
- External functions may have the new attribute `"builtin"`, additionally to `"C"` or `"FORTRAN 77"`. The `"builtin"` specification is only used for functions that are defined to be built-in in the Modelica language. The external-function call mechanism for `"builtin"` functions is implementation-defined.

The language changes are backward compatible.

D.3 Modelica 2.1

Modelica 2.1 was released January 30, 2004. The Modelica 2.1 specification was edited by Hans Olsson and Martin Otter.

D.3.1 Contributors to the Modelica Language, Version 2.1

Mikael Adlers, MathCore, Linköping, Sweden
Peter Aronsson, Linköping University, Sweden
Bernhard Bachmann, University of Applied Sciences, Bielefeld, Germany
Peter Bunus, Linköping University, Sweden
Jonas Eborn, United Technologies Research Center, Hartford, U.S.A.
Hilding Elmqvist, Dynasim, Lund, Sweden
Rüdiger Franke, ABB Corporate Research, Ladenburg, Germany
Peter Fritzson, Linköping University, Sweden
Anton Haumer, Technical Consulting & Electrical Engineering, St.Andrae-Woertern, Austria
Olof Johansson, Linköping University, Sweden
Karin Lunde, R.O.S.E. Informatik GmbH, Heidenheim, Germany
Sven Erik Mattsson, Dynasim, Lund, Sweden
Hans Olsson, Dynasim, Lund, Sweden
Martin Otter, German Aerospace Center, Oberpfaffenhofen, Germany
Levon Saldamli, Linköping University, Sweden
Christian Schweiger, German Aerospace Center, Oberpfaffenhofen, Germany
Michael Tiller, Ford Motor Company, Dearborn, MI, U.S.A.
Hubertus Tummescheit, United Technologies Research Center, Hartford, U.S.A.
Hans-Jürg Wiesmann, ABB Switzerland Ltd., Corporate Research, Baden, Switzerland

D.3.2 Main Changes in Modelica 2.1

The main changes in Modelica 2.1 are:

- Arrays and array indices of Enumerations (needed, e.g., in the `Electrical.Digital` library currently under development).
- Connections into hierarchical connectors (needed, e.g., for convenient implementation of buses).
- Optional output arguments of Modelica functions. The presence of actual input and/or output arguments can be inquired with the new built-in function `isPresent(..)`. The previous built-in function and attribute `enable` was removed.
- Making the default constraining type more useful by inheriting the base constraining type automatically to modifications.
- Enhanced redeclaration as needed, e.g., in the `Modelica.Media` library under development (e.g. `"redeclare model name"` or `"model extends name (<modifications>)"`).

- Handling of overdetermined connectors (needed, e.g., for multi-body systems and electrical power systems) including the new built-in package `Connections` with operators `Connections.branch`, `Connections.root`, `Connections.potentialRoot`, `Connections.isRoot`.
- Statement `break` in the `while` loop of an algorithm section.
- Statement `return` in a Modelica function.
- Built-in function `String(..)` to provide a string representation of `Boolean`, `Integer`, `Real` and `Enumeration` types.
- Built-in function `Integer(..)` to provide the `Integer` representation of an `Enumeration` type.
- Built-in function `semiLinear(..)` to define a characteristics with two slopes and a set of rules for symbolic transformations, especially when the function becomes underdetermined (this function is used in the Modelica Fluid library under development to define reversing flow in a mathematically clean way).
- More general identifiers by having any character in single quotes, e.g. `'+'` or `'123.456#1'` are valid identifiers. `'x'` and `x` are different identifiers. This is useful for a direct mapping of product identifiers to model names and for having the usual symbols for digital electrical signals as enumerations (such as `'+'`, `'-'`, `'0'`, `'1'`).
- New annotations:
 - For version handling of libraries and models (`version`, `uses`, `conversion`),
 - for revision logging (`revisions`),
 - for using a Modelica name as link in a HTML documentation text,
 - for convenient “inner” declaration in a GUI (`defaultComponentName`, `defaultComponentPrefixes`),
 - for parameter menu structuring (`Dialog`, `enable`, `tab`, `group`), and
 - for library specific error messages (`missingInnerMessage`, `unassignedMessage`).

Fixing some minor errors in the grammar and semantic specification.

The language changes are backward compatible, except for the introduction of the new keywords `break` and `return`, the new built-in package `Connections` and the removing of built-in function and attribute `enable`.

D.4 Modelica 2.0

Modelica 2.0 was released January, 30 2002, and the draft was released on December 18 in 2001. The Modelica 2.0 specification was edited by Hans Olsson. Modelica is a registered trademark owned by the Modelica Association since November 2001.

D.4.1 Contributors to the Modelica Language, Version 2.0

Peter Aronsson, Linköping University, Sweden
 Bernhard Bachmann, University of Applied Sciences, Bielefeld
 Peter Beater, University of Paderborn, Germany
 Dag Brück, Dynasim, Lund, Sweden
 Peter Bunus, Linköping University, Sweden
 Hilding Elmqvist, Dynasim, Lund, Sweden
 Vadim Engelson, Linköping University, Sweden
 Peter Fritzson, Linköping University, Sweden
 Rüdiger Franke, ABB Corporate Research, Ladenburg
 Pavel Grozman, Equa, Stockholm, Sweden
 Johan Gunnarsson, MathCore, Linköping
 Mats Jirstrand, MathCore, Linköping
 Sven Erik Mattsson, Dynasim, Lund, Sweden

Hans Olsson, Dynasim, Lund, Sweden

Martin Otter, German Aerospace Center, Oberpfaffenhofen, Germany

Levon Saldamli, Linköping University, Sweden

Michael Tiller, Ford Motor Company, Dearborn, MI, U.S.A.

Hubertus Tummescheit, Lund Institute of Technology, Sweden

Hans-Jürg Wiesmann, ABB Switzerland Ltd., Corporate Research, Baden, Switzerland

D.4.2 Main Changes in Modelica 2.0

A detailed description of the enhancements introduced by Modelica 2.0 is given in the papers

- M. Otter, H. Olsson: New Features in Modelica 2.0. 2nd International Modelica Conference, March 18-19, DLR Oberpfaffenhofen, Proceedings, pp. 7.1 - 7.12, 2002. This paper can be downloaded from http://www.Modelica.org/Conference2002/papers/p01_Otter.pdf
- Mattsson S. E., Elmqvist H., Otter M., and Olsson H.: Initialization of Hybrid Differential-Algebraic Equations in Modelica 2.0. 2nd International Modelica Conference, March 18-19, DLR Oberpfaffenhofen, Proceedings, pp. 9 - 15, 2002. This paper can be downloaded from http://www.Modelica.org/Conference2002/papers/p02_Mattsson.pdf

The main changes in Modelica 2.0 are:

- Full specification of initialization in order to compute consistent initial values of all variables appearing in a model before performing an operation, such as simulation or linearization.
- Specified the graphical appearance of Modelica object diagrams, thereby ensuring portability of model topology information and improving the previous informal graphical description, e.g., with separate icon and diagram positions.
- Enumeration types to allow the definition of options and properties in an understandable, safe and efficient way.
- Support for (optional) explicit preference in state-selection in order that a modeler can incorporate application specific knowledge to guide the solution process, e.g., for real-time simulation.
- Iterators in array constructors and reduction operators, to support more powerful expressions, especially in declarations, in order to avoid inconvenient and less efficient local function definitions.
- Support for generic formulation of blocks applicable to both scalar and vector connectors, connection of (automatically) vectorized blocks, and simpler input/output connectors. This allows significant simplifications of the input/output block library of Modelica, e.g., since only scalar versions of all blocks have to be provided. Furthermore, new library components can be incorporated more easily.
- Record constructor to allow, e.g., the construction of data sheet libraries.
- Functions with mixed positional and named arguments. Optional results and default arguments make the same function fit for beginners and expert users.
- Additional utilities for external C-functions that are interfaced to Modelica models, especially supporting external functions returning strings and external functions with internal memory (e.g., to interface user-defined tables, property databases, sparse matrix handling, hardware interfaces).
- Added an index, and specification of some basic constructs that had previously not formally be defined, such as while-clauses, if-clauses.

The language changes are backward compatible, except for the introduction of the new keyword enumeration and the removal of the operator `analysisType()`. The library change of the block library which will become available soon requires changes in user-models.

D.5 Modelica 1.4

Modelica 1.4 was released December 15, 2000. The Modelica Association was formed in Feb. 5, 2000 and is now responsible for the design of the Modelica language. The Modelica 1.4 specification was edited by Hans Olsson and Dag Brück.

D.5.1 Contributors to the Modelica Language, Version 1.4

Bernhard Bachmann, Fachhochschule Bielefeld, Germany
 Peter Bunus, MathCore, Linköping, Sweden
 Dag Brück, Dynasim, Lund, Sweden
 Hilding Elmqvist, Dynasim, Lund, Sweden
 Vadim Engelson, Linköping University, Sweden
 Jorge Ferreira, University of Aveiro, Portugal
 Peter Fritzson, Linköping University, Linköping, Sweden
 Pavel Grozman, Equa, Stockholm, Sweden
 Johan Gunnarsson, MathCore, Linköping, Sweden
 Mats Jirstrand, MathCore, Linköping, Sweden
 Clemens Klein-Robbenhaar, Germany
 Pontus Lidman, MathCore, Linköping, Sweden
 Sven Erik Mattsson, Dynasim, Lund, Sweden
 Hans Olsson, Dynasim, Lund, Sweden
 Martin Otter, German Aerospace Center, Oberpfaffenhofen, Germany
 Tommy Persson, Linköping University, Sweden
 Levon Saldamli, Linköping University, Sweden
 André Schneider, Fraunhofer Institute for Integrated Circuits, Dresden, Germany
 Michael Tiller, Ford Motor Company, Dearborn, MI, U.S.A.
 Hubertus Tummescheit, Lund Institute of Technology, Sweden
 Hans-Jürg Wiesmann, ABB Corporate Research Ltd., Baden, Switzerland

D.5.2 Contributors to the Modelica Standard Library

Peter Beater, University of Paderborn, Germany
 Christoph Clauß, Fraunhofer Institute for Integrated Circuits, Dresden, Germany
 Martin Otter, German Aerospace Center, Oberpfaffenhofen, Germany
 André Schneider, Fraunhofer Institute for Integrated Circuits, Dresden, Germany
 Hubertus Tummescheit, Lund Institute of Technology, Sweden

D.5.3 Main Changes in Modelica 1.4

- Removed declare-before-use rule. This simplifies graphical user environments, because there exists no order of declarations when components are graphically composed together.
- Refined package concept by introducing encapsulated classes and import mechanism. Encapsulated classes can be seen as "self-contained units": When copying or moving an encapsulated class, at most the import statements in this class have to be changed.
- Refined when-clause: The `nondiscrete` keyword is removed, equations in when-clauses must have a unique variable name on left hand side variable and the exact mapping of when-clauses to equations is defined. As a result, when-clauses are now precisely defined without referring to a sorting algorithm and it is possible to handle algebraic loops between when-clauses with different conditions and between when-clauses and the continuous-time part of a model. The `discrete` keyword is now optional, simplifying the library development because only one type of connector is needed and not several types which do contain or

do not contain the discrete prefix on variables. Additionally, when-clauses in algorithm sections may have `elsewhen`-clauses which simplifies the definition of priorities between when-clauses.

- For replaceable declarations: allowed constraining clauses, and annotations listing suitable redeclarations. This allows a graphical user environment to automatically build menus with meaningful choices.
- Functions can specify their derivative. This allows, e.g., the application of the Pantelides algorithm to reduce the index of a DAE also for external functions.
- New built-in operator "rem" (remainder) and the built-in operators `div`, `mod`, `ceil`, `floor`, `integer`, previously only allowed to be used in when-clauses can now be used everywhere, because state events are automatically generated when the result value of one of these operator changes discontinuously.
- Quantity attribute also for base types `Boolean`, `Integer`, `String` (and not only for `Real`), in order to allow abstracted variables to refer to physical quantities (e.g. `Boolean i(quantity="Current")` is true if current is flowing and is false if no current is flowing).
- `final` keyword also allowed in declaration, to prevent modification. Example:

```
model A
  Real x[:];
  final Integer n=size(x,1);
end A;
```
- Several minor enhancements, such as usage of dot-notation in modifications (e.g.: "`A x (B.C=1, B.D=2)`" is the same as "`A x (B (C=1, D=2)) ;`").
- Internally restructured specification.

Modelica 1.4 is backwards compatible with Modelica 1.3, with the exception of (1) some exotic cases where different results are achieved with the removed "declare-before-use-rule" and the previous declaration order, (2) when-clauses in equations sections, which use the general form "`expr1 = expr2`" (now only "`v=expr`" is allowed + some special cases for functions), (3) some exotic cases where a when-clause may be no longer evaluated at the initial time, because the initialization of the when-condition is now defined in a more meaningful way (before Modelica 1.4, every condition in a when-clause has a "previous" value of false), and (4) models containing the `nondiscrete` keyword which was removed.

D.6 Modelica 1.3 and Older Versions.

Modelica 1.3 was released December 15, 1999.

D.6.1 Contributors up to Modelica 1.3

The following list contributors and their affiliations at the time when Modelica 1.3 was released.

Hilding Elmqvist, Dynasim AB, Lund, Sweden

Bernhard Bachmann, ABB Corporate Research Center Heidelberg

Francois Boudaud, Gaz de France, Paris, France

Jan Broenink, University of Twente, Enschede, Netherlands

Dag Brück, Dynasim AB, Lund, Sweden

Thilo Ernst, GMD FIRST, Berlin, Germany

Ruediger Franke, ABB Network Partner Ltd. Baden, Switzerland

Peter Fritzson, Linköping University, Sweden

Alexandre Jeandel, Gaz de France, Paris, France

Pavel Grozman, Bris Data AB, Stockholm, Sweden

Kaj Juslin, VTT, Espoo, Finland

David Kågedal, Linköping University, Sweden

Mattias Klose, Technical University of Berlin, Germany

Nathalie Loubere, Gaz de France, Paris, France

Sven-Erik Mattsson, Dynasim AB, Lund, Sweden

Peter J. Mosterman, DLR Oberpfaffenhofen, Germany
 Henrik Nilsson, Linköping University, Sweden
 Hans Olsson, Dynasim AB, Lund, Sweden
 Martin Otter, DLR Oberpfaffenhofen, Germany
 Per Sahlin, Bris Data AB, Stockholm, Sweden
 Andrée Schneider, Fraunhofer Institute for Integrated Circuits, Dresden, Germany
 Michael Tiller, Ford Motor Company, Dearborn, MI, U.S.A.
 Hubertus Tummescheit, Lund Institute of Technology, Sweden
 Hans Vangheluwe, University of Gent, Belgium

D.6.2 Main Changes in Modelica 1.3

Modelica 1.3 was released December 15, 1999.

- Defined connection semantics for inner/outer connectors.
- Defined semantics for protected element.
- Defined that least variable variability prefix wins.
- Improved semantic definition of array expressions.
- Defined scope of for-loop variables.

D.6.3 Main Changes in Modelica 1.2

Modelica 1.2 was released June 15, 1999.

- Changed the external function interface to give greater flexibility.
- Introduced **inner/outer** for dynamic types.
- Redefined **final** keyword to only restrict further modification.
- Restricted redeclaration to replaceable elements.
- Defined semantics for if-clauses.
- Defined allowed code optimizations.
- Refined the semantics of event-handling.
- Introduced `fixed` and `nominal` attributes.
- Introduced `terminate` and `analysisType`.

D.6.4 Main Changes in Modelica 1.1

Modelica 1.1 was released in December 1998.

Major changes:

- Specification as a separate document from the rationale.
- Introduced prefixes discrete and nondiscrete.
- Introduced pre and when.
- Defined semantics for array expressions.
- Introduced built-in functions and operators (only connect was present in Modelica 1.0).

D.6.5 Modelica 1.0

Modelica 1, the first version of Modelica, was released in September 1997, and had the language specification as a short appendix to the rationale.

Index

- abs, 20
- access control, 29
- access operator, 19
- algorithm, 12
- algorithm sections, 115
- algorithms, 115
- and, 12, 16, 17
- annotation, 12
 - function, 159
- annotations
 - function derivative, 159
- arithmetic operators, 17
- array, 99, 101, 102, 127
 - dimension, 102
 - indexing with boolean values, 108
 - indexing with enumeration values, 108
 - reduction, 103
- array dimension
 - index bounds, 101
- assert function, 12, 78, 120
- assignment, 116
- balammodel
 - balanced, 37
- balanced models, 37
- block, 12
 - specialized class, 37
- Boolean, 13, 43
- break, 118
- built-in
 - array functions, 101
 - functions, 128
 - operators, 15
 - time variable, 19
 - types, 15, 29, 54
- cardinality, 22, 123
- ceil, 21
- change, 24
- change function, 123
- character set, 11
- class
 - base class restrictions, 62
 - declaration, 34
 - definition, 29
 - enclosing, 47
 - nested, 36
 - restricted and enhanced, 36
 - short definition, 35
 - specialized, 36
- class extends, 67
- class type, 55
- comments, 11
- component, 10
 - conditional declaration, 33
 - declaration, 29
- conditional declaration, 33
- connect, 12
- connect equation, 75
- connection
 - equations, 94
 - graphs. *See* Connection Graphs, 92
- connections, 85
- connector, 12
 - specialized class, 37
- connectors, 85
- constant, 12, 30, 32
- constrainedby, 12, 70
- constraining type, 70
- constructor functions, 102
- conversion functions, 102
- cross, 104
- declaration equations, 31, 73
- declarations, 9
- default connectable, 57
- delay, 22, 123
- der, 22, 123
- diagonal, 102
- discrete, 30, 32
- div, 21
- edge, 24
- edge function, 123
- element, 10
- else, 12, 16
- elseif, 12
- elsewhen*, 77
- empty function calls, 127
- encapsulated, 12
- encapsulation*, 143
- end, 12
- Enumeration, 44
- equality equations, 73
- equation, 12, 73, 74
 - categories, 73
- equation operators, 79, 92
- equations, 73
- escape codes, 13
- evaluation order, 16
- event-related operators, 24
- expressions
 - parameter, 26
- expressions, 15
 - constant, 26
- expressions
 - discrete-time, 26
- expressions
 - continuous-time, 27
- extends, 12, 61
 - class extends, 67
- external, 12
- false, 12
- fill, 102
- final, 12
- fixed, 46
- flattening, 51
 - partial, 51
- floating point numbers, 13
- floor, 21
- flow, 12

- for, 74, 116, 117
- formal parameters, 122
- fully qualified name, 144
- function, 12, 116, 121, 124
 - definition, 122
 - mathematical, 21
 - numeric, 20
 - reinit, 120
 - return, 122
 - specialized class, 37
- function declaration, 121
- functions, 115
 - assert, 12, 78, 120
 - change, 123
 - edge, 123
 - initial, 123
 - matrix*, 30, 99
 - sample, 123
 - terminal, 123
 - terminate, 79, 120
- generics, 61
- guard expressions, 16
- identifiers, 12
- identity, 102
- if, 12, 16, 75, 76, 118, 120
- if-expression, 18
- in, 12
- indexing
 - enumeration, 108
- information hiding*, 143
- inheritance, 61
 - multiple, 62
- initial, 24
- initial algorithm, 115
- initial equation, 82
- initial function, 123
- initialization, 82
 - number of equations needed, 83
- inner, 12, 50
- input, 30, 37, 123
- instance, 29
- instantiation, 10
- integer, 21
- Integer, 13, 43
- interface, 53
- intrinsic operators, 19
- iteration ranges, 74, 117
- keyword
 - algorithm, 12
 - annotation, 12
 - block, 12
 - break, 118
 - connect, 12
 - connector, 12
 - constant, 12
 - else, 12, 16
 - elseif, 12
 - elsewhen*, 77
 - encapsulated, 12
 - end, 12
 - equation, 12, 73, 74
 - extends, 12
 - external, 12
 - false, 12
 - final, 12
 - flow, 12
 - for, 74, 116, 117
 - function, 12, 116, 121, 124
 - if, 12, 16, 75, 76, 118, 120
 - inner, 12
 - input, 123
 - model, 12
 - output, 12
 - package, 12
 - parameter, 12
 - protected, 29
 - public, 12, 29
 - return, 122
 - then, 12, 16
 - true, 12
 - type, 12
 - when, 12, 75, 119
 - while, 118
 - within, 145
- keywords, 12
- linspace, 102
- literal constants, 12
- literals
 - Boolean, 13
 - Integer, 13
 - Real, 13
 - record, 14
 - String, 13
- local class definitions, 36
- loop, 12
- matrices, 99
- matrix, 102
- matrix algebra, 104
- matrix function*, 30, 99
- max, 103
- min, 103
- mod, 21
- model, 12
 - specialized class, 37
- model extends, 67
- MODELICAPATH, 145
- modification, 63
- modification environment, 64
- modification equations, 73
- modifications, 61
- multiple inheritance, 62
- name lookup, 47
- names, 12
- ndims, 102
- nested classes, 36
- noEvent, 24
- nominal, 46
- nondeclarative constructs, 123
- normal equations, 73
- not, 12, 16, 17
- notation and grammar, 10
- ones, 102
- operators, 15
 - $*$, 16, 17
 - $.$, 15
 - $/$, 16, 17
 - $[]$, 15
 - $^$, 15, 17
 - $+$, 16, 17
 - $<$, 16, 17
 - $<=$, 16, 17
 - $<>$, 17
 - $==$, 16, 17
 - $>$, 16, 17

-
- \geq , 16, 17
 - and, 12, 16, 17
 - associativity, 15
 - delay, 123
 - der, 123
 - equality, 17
 - logical, 17
 - miscellaneous, 18
 - not, 12, 16, 17
 - or, 16, 17
 - pre, 123
 - precedence, 15
 - reinit, 123
 - relational, 17
 - operators:, 17
 - or, 16, 17
 - outer, 50
 - outerProduct, 104
 - output, 12, 30, 37
 - overconstrained, 79, 92
 - overconstrained connection loop, 94
 - package, 12, 143–46
 - specialized class, 37
 - parameter, 12, 30, 32
 - plug-compatibility, 57
 - pre, 24, 123
 - prefix rules, 31
 - product, 103
 - protected, 29
 - public, 12, 29
 - real, 43
 - record
 - specialized class, 36
 - redeclaration, 61
 - Redeclaration, 67
 - reinit, 24, 78, 123
 - reinit function, 120
 - rem, 21
 - reserved words, 12
 - restricted and enhanced classes, 36
 - return, 122
 - sample, 24
 - sample function, 123
 - scalar, 102
 - scalar functions, 127
 - scoping, 47
 - ShirtSizes, 108
 - short class definition, 35
 - sign, 20
 - size, 102
 - skew, 104
 - smooth, 24
 - specialized classes, 36
 - sqrt, 20
 - start, 46
 - Statements, 115
 - StateSelect, 46
 - String, 13, 43
 - concatenation, 109
 - string concatenation, 18
 - subtype, 54
 - function, 59
 - restricted, 57
 - sum, 103
 - symmetric, 104
 - synchronous data-flow principle, 79
 - terminal, 24
 - terminal function, 123
 - terminate function, 79, 120
 - then, 12, 16
 - time, 19
 - transitively non-replaceable, 55, 63
 - true, 12
 - type, 12, 53, 54
 - constraining, 70
 - specialized class, 37
 - types
 - Boolean, 13, 43
 - built-in, 15, 29, 54
 - Enumeration, 44
 - Integer, 13, 43
 - predefined, 43
 - String, 13, 43
 - variability, 25
 - variable declarations, 29
 - vector, 102
 - vector algebra, 104
 - vectors, 99
 - when, 12, 75
 - when-equations
 - restrictions, 76
 - single-assignment rule, 77
 - when-statements, 119
 - restrictions, 119
 - while, 118
 - within, 145
 - zeros, 102