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

Language Specification

Version 3.7-dev

March 14, 2024

Modelica Association

Abstract

This document defines the Modelica[1] language, version 3.7-dev, 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 systems. It is suited for multi-domain modeling, for example, mechatronic models in robotics, automotive and aerospace applications involving mechanical, electrical, hydraulic control and state machine 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 `https://modelica.org`.

[1] *Modelica* is a registered trademark of the Modelica Association.

The contributors to this and to previous versions of this document are listed in appendix D.

Contents

Preface **6**

1 Introduction **8**
1.1 Overview of Modelica . . . 8
1.2 Scope of the Specification . . . 8
1.3 Some Definitions . . . 9
1.4 Notation . . . 10

2 Lexical Structure **11**
2.1 Character Set . . . 11
2.2 Comments . . . 11
2.3 Identifiers, Names, and Keywords . . . 12
2.4 Literals . . . 13
2.5 Operator Symbols . . . 14

3 Operators and Expressions **15**
3.1 Expressions . . . 15
3.2 Operator Precedence and Associativity . . . 15
3.3 Evaluation Order . . . 17
3.4 Arithmetic Operators . . . 18
3.5 Equality, Relational, and Logical Operators . . . 18
3.6 Miscellaneous Operators and Variables . . . 19
3.7 Built-in Intrinsic Operators with Function Syntax . . . 20
3.8 Variability of Expressions . . . 34

4 Classes, Predefined Types, and Declarations **39**
4.1 Access Control – Public and Protected Elements . . . 39
4.2 Double Declaration not Allowed . . . 40
4.3 Declaration Order . . . 40
4.4 Component Declarations . . . 40
4.5 Component Variability . . . 45
4.6 Class Declarations . . . 49
4.7 Specialized Classes . . . 53
4.8 Balanced Models . . . 54
4.9 Predefined Types and Classes . . . 61

5 Scoping, Name Lookup, and Flattening **68**
5.1 Flattening Context . . . 68
5.2 Enclosing Classes . . . 68
5.3 Static Name Lookup . . . 68
5.4 Inner Declarations - Instance Hierarchy Name Lookup . . . 70
5.5 Simultaneous Inner/Outer Declarations . . . 73
5.6 Flattening Process . . . 73

6 Interface or Type Relationships **79**
6.1 Interface Terminology . . . 79
6.2 The Concepts of Type, Interface and Subtype . . . 80
6.3 Interface or Type . . . 81

6.4 Interface Compatibility or Subtyping . . . 83
6.5 Plug-Compatibility or Restricted Subtyping . . . 84
6.6 Function-Compatibility or Function-Subtyping for Functions . . . 85
6.7 Type Compatible Expressions . . . 87

7 Inheritance, Modification, and Redeclaration 89
7.1 Inheritance – Extends Clause . . . 89
7.2 Modifications . . . 92
7.3 Redeclaration . . . 98
7.4 Selective Model Extension . . . 106

8 Equations 109
8.1 Equation Categories . . . 109
8.2 Flattening and Lookup in Equations . . . 109
8.3 Equations in Equation Sections . . . 109
8.4 Synchronous Data-Flow Principle and Single Assignment Rule . . . 117
8.5 Events and Synchronization . . . 118
8.6 Initialization, initial equation, and initial algorithm . . . 120

9 Connectors and Connections 126
9.1 Connect-Equations and Connectors . . . 126
9.2 Generation of Connection Equations . . . 132
9.3 Restrictions of Connections and Connectors . . . 135
9.4 Overconstrained Connections . . . 137

10 Arrays 144
10.1 Array Declarations . . . 144
10.2 Flexible Array Sizes . . . 147
10.3 Built-in Array Functions . . . 147
10.4 Vector, Matrix and Array Constructors . . . 152
10.5 Indexing . . . 156
10.6 Scalar, Vector, Matrix, and Array Operator Functions . . . 157
10.7 Empty Arrays . . . 162

11 Statements and Algorithm Sections 164
11.1 Algorithm Sections . . . 164
11.2 Statements . . . 165

12 Functions 172
12.1 Function Declaration . . . 172
12.2 Function as a Specialized Class . . . 174
12.3 Pure Modelica Functions . . . 175
12.4 Function Call . . . 177
12.5 Built-in Functions . . . 184
12.6 Record Constructor Functions . . . 184
12.7 Derivatives and Inverses of Functions . . . 188
12.8 Function Inlining and Event Generation . . . 197
12.9 External Function Interface . . . 199

13 Packages 214
13.1 Package as Specialized Class . . . 214
13.2 Importing Definitions from a Package . . . 214
13.3 The Modelica Library Path – MODELICAPATH . . . 216
13.4 File System Mapping of Package/Class . . . 217
13.5 External Resources . . . 218
13.6 Multilingual Descriptions . . . 219

14 Overloaded Operators 222
14.1 Overview of Overloaded Operators . . . 222

14.2 Matching Function . . . 223
14.3 Overloaded Constructors . . . 223
14.4 Overloaded String Conversions . . . 224
14.5 Overloaded Binary Operations . . . 224
14.6 Overloaded Unary Operations . . . 225
14.7 Example of Overloading for Complex Numbers . . . 226

15 Stream Connectors **230**
15.1 Definition of Stream Connectors . . . 230
15.2 inStream and Connection Equations . . . 231
15.3 actualStream . . . 235

16 Synchronous Language Elements **236**
16.1 Rationale for Clocked Semantics . . . 237
16.2 Definitions . . . 238
16.3 Clock Constructors . . . 241
16.4 Clocked State Variables . . . 244
16.5 Partitioning Operators . . . 244
16.6 Clocked When-Clause . . . 248
16.7 Clock Partitioning . . . 249
16.8 Discretized Sub-Partition . . . 252
16.9 Initialization of Clocked Partitions . . . 257
16.10 Other Operators . . . 257
16.11 Semantics . . . 258

17 State Machines **260**
17.1 Transitions . . . 260
17.2 State Machine Graphics . . . 262
17.3 State Machine Semantics . . . 263

18 Annotations **272**
18.1 Notation for Annotation Definitions . . . 272
18.2 Vendor-Specific Annotations . . . 272
18.3 Documentation . . . 273
18.4 Symbolic Processing . . . 278
18.5 Simulations . . . 279
18.6 Usage Restrictions . . . 281
18.7 Graphical Objects . . . 282
18.8 Graphical User Interface . . . 292
18.9 Versions . . . 297
18.10 Access Control to Protect Intellectual Property . . . 303
18.11 Functions . . . 308
18.12 Choices for Modifications and Redeclarations . . . 308

19 Unit Expressions **309**
19.1 The Syntax of Unit Expressions . . . 309

20 The Modelica Standard Library **311**

A Modelica Concrete Syntax **312**
A.1 Lexical conventions . . . 312
A.2 Grammar . . . 313

B Modelica DAE Representation **320**

C Derivation of Stream Equations **324**

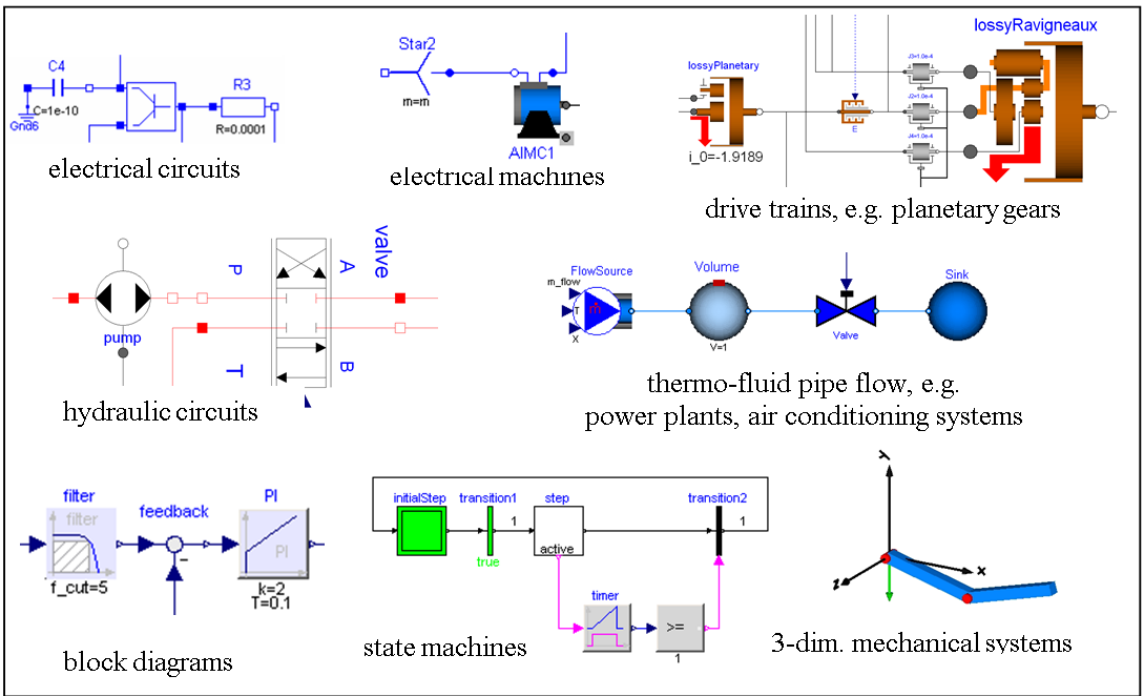
D Modelica Revision History **329**

Bibliography **330**

Index **331**

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 below:



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 `https://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 `https://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 text directly under the chapter headings are non-normative introductions to the chapters.

The Modelica language has been developed since 1996. This document describes version 3.7-dev of the Modelica language. The revision history is available in appendix D.

Chapter 1

Introduction

1.1 Overview of Modelica

Modelica is a language for modeling of cyber-physical systems, supporting acausal connection of components governed by mathematical equations to facilitate modeling from first principles. It provides object-oriented constructs that facilitate reuse of models, and can be used conveniently for modeling complex systems containing, e.g., mechanical, electrical, electronic, magnetic, hydraulic, thermal, control, electric power or process-oriented subcomponents.

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. The semantic specification should be read together with the Modelica grammar.

A class (of specialized class **`model`** or **`block`**) intended to be simulated on its own is called a *simulation model*.

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 for those classes, before using them to build the simulation model.

There are specific semantic restrictions for a simulation model to ensure that the model is complete; they allow its flat Modelica structure to be further transformed into a set of differential, algebraic and discrete equations (= flat hybrid DAE). Note that satisfying the semantic restrictions does not guarantee that the model can be initialized from the initial conditions and simulated.

Modelica was designed to facilitate symbolic transformations of models, especially by mapping basically every Modelica language construct to 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. In order to allow this structural analysis, a tool may reject simulating a model if parameters cannot be evaluated during translation – due to calls of external functions or initial equations/initial algorithms for `fixed = false` parameters. Accepting such models is a quality of implementation issue. 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).
- The **when**-clauses where every **when**-clause is treated as a set of conditionally evaluated 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. Initial setup of the model is specified using `start`-attributes and equations that hold only during initialization.

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

Explanations of many terms can be found using the document index in appendix D. Some important terms are defined below.

Definition 1.1. ***Component.*** An element defined by the production *`component-clause`* in the Modelica grammar (basically a variable or an instance of a class) □

Definition 1.2. ***Element.*** Class definition, **extends**-clause, or *`component-clause`* declared in a class (basically a class reference or a component in a declaration). □

Definition 1.3. ***Flattening.*** The translation of a model described in Modelica to the corresponding model described as a hybrid DAE (see appendix B), involving expansion of inherited base classes, parameterization of base classes, local classes and components, and generation of connection equations from `connect`-equations. In other words, 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. □

Definition 1.4. ***Initialization.*** Simulation starts with solving the initialization problem at the starting time, resulting in values for all variables that are consistent with the result of the flattening. □

Definition 1.5. ***Transient analysis.*** Starting from the result of the initialization problem, the model is simulated forward in time. This uses numerical methods for handling the hybrid DAE, resulting in solution trajectories for the model's variables, i.e., the value of the variables as a function of time. □

[*In the numerical literature transient analysis is often called solving the* initial value problem*, but that term is not used here to avoid confusion with the initialization problem.*]

Definition 1.6. ***Simulation.*** Simulation is the combination of initialization followed by transient analysis. □

[*The model can be analyzed in ways other than simulation, e.g., linearization, and parameter estimation, but they are not described in the specification.*]

Definition 1.7. ***Translation.*** Translation is the process of preparing a Modelica simulation model for simulation, starting with flattening but not including the simulation itself. □

[*Typically, in addition to flattening, translation involves symbolic manipulation of the hybrid DAE and transforming the result into computer code that can simulate the model.*]

1.4 Notation

The remainder of this section shows examples of the presentation used in this document.

Syntax highlighting of Modelica code is illustrated by the code listing below. Things to note include keywords that define code structure such as **equation**, keywords that do not define code structure such as connect, and recognized identifiers with meaning defined by the specification such as semiLinear:

```
model Example "Example used to illustrate syntax highlighting"
  /* The string above is a class description string, this is a comment. */
  /* Invalid code is typically presented like this: */
  String s = 1.0; // Error: No conversion form Real to String.
  Real x;
equation
  2 * x = semiLinear(time - 0.5, 2, 3);
  /* The annotation below has omitted details represented by an ellipsis: */
  connect(resistor.n, conductor.p) annotation(...);
end Example;
```

Relying on implicit conversion of Integer literals to Real is common, as seen in the equation above (note use of Modelica code appearing inline in the text).

It is common to mix Modelica code with mathematical notation. For example, average(x, y) could be defined as $\frac{x+y}{2}$.

Inline code fragments are sometimes surrounded by quotes to clearly mark their beginning and end, or to emphasize separation from the surrounding text. For example, ‘,’ is used to separate the arguments of a function call.

Definition 1.8. ***Something.*** Text defining the meaning of *something*. □

In addition to the style of definition above, new terminology can be introduced in the running text. For example, a *dummy* is something that...

[*This is non-normative content that provides some explanation, motivation, and/or additional things to keep in mind. It has no defining power and may be skipped by readers strictly interested in just the definition of the Modelica language.*]

[*Example: This is an example, which is a special kind of non-normative content. Examples often contain a mix of code listings and explanatory text, and this is no exception:*

```
String s = 1.0; // Error: No conversion form Real to String.
```

To fix the type mismatch above, the number has to be replaced by a String *expression, such as* "1.0".]

Other code listings in the document include specification of lexical units and grammatical structure, both using metasymbols of the extended BNF-grammar defined in appendix A.1. Lexical units are named with all upper-case letters and introduced with the ‘=’ sign:

```
SOME-TOKEN = NON-DIGIT { DIGIT | NON-DIGIT }
```

Grammatical structure is recognized by production rules being named with lower-case letters and introduced with the ‘:’ sign (also note appearance of the Modelica keyword der):

```
differentiated-expression :
    der "(" SOME-TOKEN ")"
    | "(" differentiated-expression "+" differentiated-expression ")"
```

Annotations are defined using the syntactic forms of Modelica record definitions and component declarations, but with special semantics given in section 18.1.

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 literals, 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 A.

2.1 Character Set

The character set of the Modelica language is Unicode, but restricted to the Unicode characters corresponding to 7-bit ASCII characters for identifiers; see appendix A.1.

2.2 Comments

There are two kinds of comments in Modelica which are not lexical units in the language and therefore are treated as white-space by a Modelica translator. The white-space characters are space, tabulator, and line separators (carriage return and line feed); and white-space cannot occur inside tokens, e.g., <= must be written as two characters without space or comments between them. The following comment variants are available:

```
// Rest-of-line comment: Everything from // to the end of the line are ignored.
"Not part of comment"
/* Delimited comment: Characters after /* are ignored,
  including line termination. The comment ends with */
```

[*The comment syntax is identical to that of C++.*]

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

```
/* Invalid nesting of comments, the comment ends just before 'end'
model Interesting
  /* To be done */
end Interesting;
*/
```

Rest-of-line comments can safely be used to comment out blocks of code without risk of conflict with comments inside.

```
//model Valid // Some other comment
//  /* To be done */
//end Valid;
```

There is also a description-string, that is part of the Modelica language and therefore not ignored by the Modelica translator. Such a description-string may occur at the end of a declaration, equation, or statement or at the beginning of a class definition. 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 starts with a letter or underscore ('_'), followed by any number of letters, digits, or underscores. Case is significant, i.e., the identifiers `Inductor` and `inductor` are different. The second form (*Q-IDENT*) starts with a single quote, followed by a sequence of any printable ASCII character, where single-quote must be preceded by backslash, and terminated by a single quote, e.g., `'12H'`, `'13\'H'`, `'+foo'`. Control characters in quoted identifiers have to use string escapes. The single quotes are part of the identifier, i.e., `'x'` and `x` are distinct identifiers. The redundant escapes (`'\?'` and `'\"'`) are the same as the corresponding non-escaped variants (`'?'` and `'"'`), but are only for use in Modelica source code. A full BNF definition of the Modelica syntax and lexical units is available in appendix A.

```
IDENT = NON-DIGIT { DIGIT | NON-DIGIT } | Q-IDENT
Q-IDENT = "'" { Q-CHAR | S-ESCAPE } "'"
NON-DIGIT = "_" | letters "a" ... "z" | letters "A" ... "Z"
DIGIT = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
Q-CHAR = NON-DIGIT | DIGIT | "!" | "#" | "$" | "%" | "&" | "(" | ")"
   | "*" | "+" | "," | "-" | "." | "/" | ":" | ";" | "<" | ">" | "="
   | "?" | "@" | "[" | "]" | "^" | "{" | "}" | "|" | "~" | " " | """
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.

[*Example: A name:* `Ele.Resistor`]

A *component reference* is 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 (section 4.4), or expression for the case of a local iterator variable (section 10.6.9).

[*Example: A component reference:* `Ele.Resistor.u[21].r`]

2.3.3 Modelica Keywords

The following Modelica *keywords* are reserved words that cannot be used where *IDENT* is expected in the language grammar (appendix A):

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

In particular, it is not allowed to declare an element or enumeration literal with these names. This also applies to the identifiers that name the predefined types `Real`, `Integer`, `Boolean`, and `String`, see section 4.9.

[*Example: Not all predefined types have names with restrictions:*

```
type StateSelect = enumeration(one, two);
StateSelect s = StateSelect.one;           // OK, using local StateSelect.
Real x(stateSelect = StateSelect.never);   // Error: 'never' is not a literal
                                           // of StateSelect.
Real y(stateSelect = .StateSelect.never); // OK, using predefined StateSelect.
```

]

2.4 Literals

Literals are unnamed constants used to build expressions, and 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 followed by a decimal point, followed by decimal digits, followed by an exponent indicated by `E` or `e` followed by a sign and one or more decimal digits. The various parts can be omitted, see ***UNSIGNED-REAL*** in appendix A.1 for details and also the examples below. The minimal recommended range is that of IEEE double precision floating point numbers, for which the largest representable positive number is $1.7976931348623157 \times 10^{308}$ and the smallest positive number is $2.2250738585072014 \times 10^{-308}$. For example, the following are floating point number literals:

```
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, 0.13E2, .13E2
```

The last variant shows that that the leading zero is optional (in that case decimal digits must be present). Note that `13` is not in this list, since it is not a floating point number, but can be converted to a floating point number.

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`. The range of supported `Integer` literals shall be at least large enough to represent the largest positive `IntegerType` value, see section 4.9.2.

[*Negative numbers are formed by unary minus followed by an integer literal.*]

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 (see appendix A.1 for allowed characters) apart from double quote (") and backslash (\ ), including new-line, can be *directly* included in a string without using an escape sequence. Certain characters in string literals can be represented using escape sequences, i.e., the character is preceded by a backslash (\) within the string. Those characters are:

Character	*Description*
`\'`	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`	Newline (code 10, ctrl-J), same as literal newline
`\r`	Carriage 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.

The `"\n"` character is used to conceptually indicate the end of a line within a Modelica string. Any Modelica program that needs to recognize line endings can check for a single `"\n"` character to do so on any platform. It is the responsibility of a Modelica implementation to make any necessary transformations to other representations when writing to or reading from a text file.

[*For example, a* `"\n"` *is written and read as-is in a Unix or Linux implementation, but written as* `"\r\n"` *pair, and converted back to* `"\n"` *when read in a Windows implementation.*]

[*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 312 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 A. For example, they can be built from operators, function references, components, or component references (referring to components) and literals. Each expression has a type and a variability.

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 4.9.

3.1 Expressions

Modelica equations, assignments and declaration equations contain expressions.

Expressions can contain basic operations, `+`, `-`, `*`, `/`, `^`, etc. with normal precedence as defined in table 3.1 in section 3.2 and the grammar in appendix A. 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 12.4.4. 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 implicit subexpression structure of expressions with operators. (Explicit subexpression structure can be expressed by wrapping the subexpression in parentheses.) An operator with higher precedence ties harder to its operands than an operator with lower precedence. For example, '`*`' having higher precedence than '`+`' means that `1 + 2 * 3` is implicitly structured as `1 + (2 * 3)`.

Precedence group associativity is used to determine the implicit subexpression structure when operators belong to the same group of equal precedence. Left associativity means that subexpressions are formed from left to right. For example, left associativity of binary additive operators means that `1 - 2 - 3` is implicitly structured as `(1 - 2) - 3`. A precedence group may also be non-associative, meaning that there is no implicit subexpression structure defined based on associativity. For example, non-associativity of relational operators means that `1 < 2 < 3` is an invalid expression. Note that the operators don't need to be identical for associativity to matter; also `1 == 2 < 3` is invalid, and `1 - 2 + 3` is implicitly structured as `(1 - 2) + 3`. Also note that the non-associative array range in Modelica can be used with either two or three operands separated by '`:`', meaning that `1 : 2 : 5` is one valid ternary use of the operator rather than two invalid binary uses of the operator.

At the parsing stage – which is where the here defined operator precedence and associativity matters – the subexpression structure is fixed. Since Modelica tools have the freedom to symbolically manipulate expressions, this subexpression structure cannot be expected to reflect order of evaluation, compare section 3.3.

The following table presents the precedence and associativity of all the expression operators, consistent with and complementing information that can be derived from the Modelica grammar in appendix A.

Table 3.1: *Operators in order of precedence from highest to lowest. Operators with different precedence are separated by horizontal lines. All operators are binary except 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.*
† *The associativity of array construction and concatenation refers to the separator (‘,’ or ‘;’), not the enclosing delimiters.*

Operator group	*Assoc.*	*Operator syntax*	*Examples*
Postfix array index	left	`[]`	`arr[index]`
Postfix access	left	`.`	`a.b`
Postfix function call	none	*funcName*`(`*args*`)`	`sin(4.36)`
Array construction	left†	`{`*expr*`,` *expr*`, ...}`	`{2, 3}`
Horizontal concatenation	left†	`[`*expr*`,` *expr*`, ...]`	`[5, 6]`
Vertical concatenation	left†	`[`*expr*`;` *expr*`; ...]`	`[2, 3; 7, 8]`
Exponentiation	none	`^`	`2 ^ 3`
Multiplicative	left	`* /`	`2 * 3, 2 / 3`
Elementwise multiplicative	left	`.* ./`	`{2, 3} .* {4, 5}`
Additive unary	none	`+`*expr* `-`*expr*	`-0.5`
Additive	left	`+ -`	`1 + 2`
Elementwise additive	left	`.+ .-`	`{2, 3} .+ {4, 5}`
Relational	none	`< <= > >= == <>`	`a < b, a <= b, a > b`
Unary negation	none	**`not`** *expr*	**`not`** `b1`
Logical and	left	**`and`**	`b1` **`and`** `b2`
Logical or	left	**`or`**	`b1` **`or`** `b2`
Array range	none	*expr* `:` *expr*	`1 : 5`
	none	*expr* `:` *expr* `:` *expr*	`start : step : stop`
Conditional	none	**`if`** *expr* **`then`** *expr* **`else`** *expr*	**`if`** `b` **`then`** `3` **`else`** `x`
Named argument	none	*ident* `=` *expr*	`x = 2.26`

The postfix array index and postfix access operators are not merely expression operators in the normal sense that `a.b[1]` can be treated as `a.(b[1])`. Instead, these operators need to be considered jointly to identify an entire ***`component-reference`*** (one of the alternative productions for ***`primary`*** in the grammar) which is the smallest unit that can be seen as an expression in itself. Postfix access can only be applied immediately to a ***`component-reference`***; not even parentheses around the left operand are allowed. Postfix array index can additionally be applied if there are parentheses around the left operand, see section 10.5.

[*Example: Relative precedence of postfix array index and postfix access. Consider the following definition of the array variable* `a`*:*

```
record R
  Real[2] x;
end R;
R[3] a;
```

These are some valid as well as invalid ways to using postfix array index and postfix access:

```
a[3].x[2]    // OK: Component reference of type Real
a[3].x       // OK: Component reference of type Real[2]
a.x[2]       // OK: Component reference of type Real[3]
a.x[2, :]    // Error.
a.x          // OK: Component reference of type Real[3, 2]
```

```
(a.x)[2]     // OK: Component reference of type Real[2] - same as a[2].x[:]
(a.x)[2, :] // OK: Component reference of type Real[2] - same as a[2].x[:]
a[3]         // OK: Component reference of type R
(a[3]).x     // Error: Invalid use of parentheses
```

The relation between `a.x`, `a.x[2]`, *and* `(a.x)[2]` *illustrates the effect of giving higher precedence to array index than postfix access. Had the precedence been equal, this would have changed the meaning of* `a.x[2]` *to the same thing that* `(a.x)[2]` *expresses, being a component reference of type* `Real[2]`.]

[*Example: Non-associative exponentiation and array range operator (note that the array range operator only takes scalar operands):*

```
x ^ y ^ z     // Not legal, use parentheses to make it clear.
a : b : c : d // Not legal, and parentheses cannot make it legal.
```

]

The additive unary expressions are only allowed in the first term of a sum, that is, not immediately to the right of any of the additive or elementwise additive operators. For example, `1 + -1 + 1` is an invalid expression (not parseable according to appendix A), whereas both `1 + (-1) + 1` and `-1 + 1 + 1` are fine.

[*Example: The unary minus and plus in Modelica is slightly different than in Mathematica*[1] *and in MATLAB*[2]*, since the following expressions are illegal (whereas in Mathematica 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
```

]

The conditional operator may also include **`elseif`**-branches.

Equality `=` and assignment `:=` are not expression operators since they are allowed only in equations and in assignment statements respectively.

[*The operator precedence table is useful when generating textual representations of Modelica expression trees. When doing this, attention must be paid to the rule that the unary additive operators are only allowed for the first term in a sum. A naive implementation might not produce all the required parentheses for an expression tree such as* `1 + (-1)`*, as it might think that the higher precedence of the unary operator makes the parentheses redundant. A trick that solves this problem is to instead treat the additive unary operators as left associative with the same precedence as the binary additive operators.*]

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 branches 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 recommended to automatically convert the number to another type with greater precision.

[*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;
```

[1] *Mathematica* is a registered trademark of Wolfram Research Inc.
[2] *MATLAB* is a registered trademark of MathWorks Inc.

```
equation
  b = (I >= 1 and I <= n) and v[I];                  // Unsafe, may result in
    error
  b = if (I >= 1 and I <= n) then v[I] else false; // Safe
```

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
```

]

3.4 Arithmetic Operators

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

Operator	*Description*
^	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-operator ] term { add-operator term }

add-operator :
   "+" | "-"

term :
   factor { mul-operator factor }

mul-operator :
   "*" | "/"

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`:

Operator	*Description*
>	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:

Operator	*Description*
`not`	Logical negation (unary operator)
`and`	Logical *and* (conjunction)
`or`	Logical *or* (disjunction)

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 [ relational-operator arithmetic-expression ]

relational-operator :
   "<" | "<=" | ">" | ">=" | "==" | "<>"
```

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 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.*]
- Relational operators can generate events, see section 3.8.5.

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.

[*Example:* `"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 6.7) giving the type of the **if**-expression. The **if**-expressions with **elseif** are defined by replacing **elseif** by **else if**. For short-circuit evaluation see section 3.3.

[**elseif** *in expressions has been added to the Modelica language for symmetry with* **if***-equations.*]

[*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 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 models and blocks, 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 3.7.4 below.
- Event-related operators with function syntax, see section 3.7.5 below.
- Array operators/functions, see section 10.1.1.

Note that when the specification references a function having the name of a built-in function it references the built-in function, not a user-defined function having the same name, see also section 12.5. With exception of the built-in `String` operator, all operators in this section can only be called with positional arguments.

3.7.1 Numeric Functions and Conversion Functions

The mathematical functions and conversion operators listed below do not generate events.

Expression	*Description*	*Details*
`abs(`v`)`	Absolute value (event-free)	Function 3.1
`sign(`v`)`	Sign of argument (event-free)	Function 3.2
`sqrt(`v`)`	Square root	Function 3.3
`Integer(`e`)`	Conversion from enumeration to `Integer`	Operator 3.1
`EnumTypeName(`i`)`	Conversion from `Integer` to enumeration	Operator 3.2
`String(...)`	Conversion to `String`	Operator 3.3

All of these except for the `String` conversion operator are vectorizable according to section 12.4.6.

Additional non-event generating mathematical functions are described in section 3.7.3, whereas the event-triggering mathematical functions are described in section 3.7.2.

Function 3.1 abs

`abs(`v`)`

Expands into `noEvent(if` v `>= 0 then` v `else -`v`)`. Argument v needs to be an `Integer` or `Real` expression.

[*By not generating events the property* `abs(`x`)` ≥ 0 *for all* x *is ensured at the cost of sometimes having a derivative that changes discontinuously between events.*

A typical case requiring the event-free semantics is a flow equation of the form `abs(x) * x = y`. *With event generation, the equation would switch between the two forms* `x^2 = y` *and* `-x^2 = y` *at the events, where the events would not be coinciding exactly with the sign changes of* `y`. *When* `y` *passes through zero, neither form of the equation would have a solution in an open neighborhood of* `y` $= 0$, *and hence solving the equation would have to fail at some point sufficiently close to* `y` $= 0$. *Without event generation, on the other hand, the equation can be solved easily for* `x`, *also as* `y` *passes through zero. Note that without event generation the derivative of* `abs(x) * x` *never changes discontinuously, despite* `abs(x)` *having a discontinuous derivative.*

In inverted form this equation is `x = sign(y) * sqrt(abs(y))`. *With event generation, the call to* `sqrt` *would fail when applied to a negative number during root finding of the zero crossing for* `abs(y)`, *compare section 8.5. Without event generation, on the other hand, evaluating* `sqrt(abs(y))` *will never fail.*]

Function 3.2 sign

`sign(`v`)`

Expands into `noEvent(if` v `> 0 then 1 else if` v `< 0 then -1 else 0)`. Argument v needs to be an `Integer` or `Real` expression.

Function 3.3 sqrt

`sqrt(`v`)`

Square root of v if $v \geq 0$, otherwise an error occurs. Argument v needs to be an `Integer` or `Real` expression.

Operator 3.1 Integer

```
Integer(e)
```

Ordinal number of the expression e of enumeration type that evaluates to the enumeration value `E.enumvalue`, where `Integer(E.e1) = 1`, `Integer(E.en) = n`, for an enumeration type `E = enumeration(e1, ..., en)`. See also section 4.9.5.2.

Operator 3.2 <EnumTypeName>

```
EnumTypeName(i)
```

For any enumeration type `EnumTypeName`, returns the enumeration value `EnumTypeName.e` such that `Integer(EnumTypeName.e)` $= i$. Refer to the definition of `Integer` above.

It is an error to attempt to convert values of i that do not correspond to values of the enumeration type. See also section 4.9.5.3.

Operator 3.3 String

```
String(b, ⟨options⟩)
String(i, ⟨options⟩)
String(i, format = s)
String(r, ⟨options⟩)
String(r, format = s)
String(e, ⟨options⟩)
```

Convert a scalar non-`String` expression to a `String` representation. The first argument may be a `Boolean` b, an `Integer` i, a `Real` r, or an enumeration value e (section 4.9.5.2). The ⟨*options*⟩ represent zero or more of the following named arguments (that cannot be passed as positional arguments):

- `Integer minimumLength = 0`: Minimum length of the resulting string. If necessary, the blank character is used to fill up unused space.
- `Boolean leftJustified = true`: If true, the converted result is left justified in the string; if false it is right justified in the string.
- `Integer significantDigits = 6`: Number of significant digits in the result string. Only allowed when formatting a `Real` value.

The standard type coercion described in section 10.6.13 shall not be applied for the first argument of `String`. Hence, specifying `significantDigits` is an error when the first argument of `String` is an `Integer` expression.

For `Real` expressions the output shall be according to the Modelica grammar.

[*Examples of* `Real` *values formatted with 6 significant digits:* 12.3456, 0.0123456, 12345600, 1.23456E-10.]

The `format` string corresponding to ⟨*options*⟩ is:

- For `Real`:
  `(if leftJustified then "-" else "") + String(minimumLength)`
  `  + "." + String(signficantDigits) + "g"`
- For `Integer`:
  `(if leftJustified then "-" else "") + String(minimumLength) + "d"`

The ANSI-C style `format` string (which cannot be combined with any of the other named arguments) consists of a single conversion specification without the leading %. It shall not contain a length modifier, and shall not use '`*`' for width and/or precision. For both `Real` and `Integer` values, the conversion specifiers '`f`', '`e`', '`E`', '`g`', '`G`' are allowed. For `Integer` values it is also allowed to use the '`d`', '`i`', '`o`', '`x`', '`X`', '`u`', and '`c`' conversion specifiers. Using the `Integer` conversion specifiers for a `Real` value is a deprecated feature, where tools are expected to produce a result by either rounding the value, truncating the value, or picking one of the `Real` conversion specifiers instead.

The 'x'/'X' formats (hexa-decimal) and c (character) for Integer values give results that do not agree with the Modelica grammar.

[*Example: Some situations worth a remark:*

- `String(4.0, format = "g")` *produces* 4 *which is not a valid* `Real` *literal. However, it is an* `Integer` *literal that can be used almost anywhere in Modelica code instead of the* `Real` *literal* `4.0` *(with the first argument to* `String` *being a notable exception here).*
- `String(4, format = ".3f")` *uses the* `Integer` *case of* `String` *since no automatic type coerction takes place for the first argument. An implementation may internally convert the value to floating point and then fall back on the* `Real` *case implementation of* `format = ".3f"`.
- `String(4611686018427387648, format = ".0f")` *(a valid* `Integer` *value in an implementation with 64 bit* `IntegerType`*) may produce* 4611686018427387904 *(not equal to input value), in case internal conversion to a 64 bit* `double` *is applied.*

]

3.7.2 Event Triggering Mathematical Functions

The operators listed below trigger events if used outside of a **when**-clause and outside of a clocked discrete-time partition (see section 16.8.1).

Expression	*Description*	*Details*
`div(`x`, `y`)`	Division with truncation toward zero	Operator 3.4
`mod(`x`, `y`)`	Integer modulus	Operator 3.5
`rem(`x`, `y`)`	Integer remainder	Operator 3.6
`ceil(`x`)`	Smallest integer `Real` not less than x	Operator 3.7
`floor(`x`)`	Largest integer `Real` not greater than x	Operator 3.8
`integer(`x`)`	Largest `Integer` not greater than x	Operator 3.9

These expression for `div`, `ceil`, `floor`, and `integer` are event generating expression. The event generating expression for `mod(x,y)` is `floor(x/y)`, and for `rem(x,y)` it is `div(x,y)` – i.e., events are not generated when `mod` or `rem` changes continuously in an interval, but when they change discontinuously from one interval to the next.

[*If this is not desired, the* `noEvent` *operator can be applied to them. E.g.,* `noEvent(integer(v))`.]

Operator 3.4 `div`

`div(`x`, `y`)`

Algebraic quotient x/y with any fractional part discarded (also known as truncation toward zero).

[*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`.

Operator 3.5 `mod`

`mod(`x`, `y`)`

Integer modulus of x/y, i.e., `mod(`x`, `y`) = `x` - 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:* `mod(3, 1.4) = 0.2`, `mod(-3, 1.4) = 1.2`, `mod(3, -1.4) = -1.2`.]

Operator 3.6 `rem`

`rem(`x`, `y`)`

Integer remainder of x/y, such that `div(`x`, `y`) * `y` + 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:* `rem(3, 1.4) = 0.2`, `rem(-3, 1.4) = -0.2`.]

Operator 3.7 `ceil`

`ceil(`x`)`

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.*]

Operator 3.8 `floor`

`floor(`x`)`

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.*]

Operator 3.9 `integer`

`integer(`x`)`

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.*]

3.7.3 Elementary Mathematical Functions

The functions listed below are elementary mathematical functions. Tools are expected to utilize well known properties of these functions (derivatives, inverses, etc) for symbolic processing of expressions and equations.

Expression	*Description*	*Details*
`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(`y`, `x`)`	Principal value of the arc tangent of y/x	Function 3.4
`sinh(`x`)`	Hyperbolic sine	
`cosh(`x`)`	Hyperbolic cosine	
`tanh(`x`)`	Hyperbolic tangent	
`exp(`x`)`	Exponential, base e	
`log(`x`)`	Natural (base e) logarithm ($x > 0$)	
`log10(`x`)`	Base 10 logarithm ($x > 0$)	

These functions are the only ones that can also be called using the deprecated `"builtin"` external language, see section 12.9.

[*End user oriented information about the elementary mathematical functions can be found for the corresponding functions in the* `Modelica.Math` *package.*]

Function 3.4 `atan2`

`atan2(`y`, `x`)`

Principal value of the arc tangent of y/x, using the signs of the two arguments to determine the quadrant of the result. The result φ is in the interval $[-\pi, \pi]$ and satisfies:

$$|(x, y)| \cos(\varphi) = x$$
$$|(x, y)| \sin(\varphi) = y$$

3.7.4 Derivative and Special Purpose Operators with Function Syntax

The operators listed below include the derivative operator and special purpose operators with function syntax.

Expression	*Description*	*Details*
`der`(*expr*)	Time derivative	Operator 3.10
`delay`(*expr*, ...)	Time delay	Operator 3.11
`cardinality`(*c*)	Number of occurrences in `connect`-equations	Operator 3.12
`homotopy`(*actual*, *simplified*)	Homotopy initialization	Operator 3.13
`semiLinear`(*x*, k^+, k^-)	Sign-dependent slope	Operator 3.14
`inStream`(*v*)	Stream variable flow into component	Operator 3.15
`actualStream`(*v*)	Actual value of stream variable	Operator 3.16
`spatialDistribution`(...)	Variable-speed transport	Operator 3.17
`getInstanceName`()	Name of instance at call site	Operator 3.18

The special purpose operators with function syntax where the call below uses named arguments can be called with named arguments (with the specified names), or with positional arguments (the inputs of the functions are in the order given in the calls below).

Operator 3.10 `der`

`der`(*expr*)

The time derivative of *expr*. If the expression *expr* is a scalar it needs to be a subtype of `Real`. The expression and all its time-varying subexpressions must be continuous and semi-differentiable. If *expr* 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.

[*For* `Real` *parameters and constants the result is a zero scalar or array of the same size as the variable.*]

Operator 3.11 `delay`

`delay`(*expr*, *delayTime*, *delayMax*)
`delay`(*expr*, *delayTime*)

Evaluates to *expr*`(time -` *delayTime*`)` for `time` > `time.start` + *delayTime* and *expr*`(time.start)` for `time` ≤ `time.start` + *delayTime*. The arguments, i.e., *expr*, *delayTime* and *delayMax*, need to be subtypes of `Real`. *delayMax* needs to be additionally a parameter expression. The following relation shall hold: $0 \leq delayTime \leq delayMax$, otherwise an error occurs. If *delayMax* is not supplied in the argument list, *delayTime* needs to be a parameter expression. For non-scalar arguments the function is vectorized according to section 10.6.12. For further details, see section 3.7.4.1.

Operator 3.12 `cardinality`

`cardinality`(*c*)

[*This is a deprecated operator. It should no longer be used, since it will be removed in one of the next Modelica releases.*]

Returns the number of (inside and outside) occurrences of connector instance *c* in a `connect`-equation as an `Integer` number. For further details, see section 3.7.4.3.

Operator 3.13 `homotopy`

`homotopy(actual =` *actual*`, simplified =` *simplified*`)`

The scalar expressions *actual* and *simplified* are subtypes of `Real`. A Modelica translator should map this operator into either of the two forms:

1. Returns *actual* (trivial implementation).
2. In order to solve algebraic systems of equations, the operator might during the solution process return a combination of the two arguments, ending at actual.

   [*Example: actual* $\cdot\ \lambda +$ *simplified* $\cdot\ (1 - \lambda)$*, where* λ *is a homotopy parameter going from 0 to 1.*]

   The solution must fulfill the equations for `homotopy` returning *actual*.

For non-scalar arguments the function is vectorized according to section 12.4.6. For further details, see section 3.7.4.4.

Operator 3.14 `semiLinear`

```
semiLinear(x, k+, k-)
```

Returns: `smooth(0, if` x `>= 0 then` k^+ `*` x `else` k^- `*` x`)`. The result is of type `Real`. For non-scalar arguments the function is vectorized according to section 10.6.12. For further details, see section 3.7.4.5 (especially in the case when $x = 0$).

Operator 3.15 `inStream`

```
inStream(v)
```

`inStream(`v`)` is only allowed for stream variables v defined in stream connectors, and is the value of the stream variable v close to the connection point assuming that the flow is from the connection point into the component. This value is computed from the stream connection equations of the flow variables and of the stream variables. The operator is vectorizable. For further details, see section 15.2.

Operator 3.16 `actualStream`

```
actualStream(v)
```

`actualStream(`v`)` returns the actual value of the stream variable v for any flow direction. The operator is vectorizable. For further details, see section 15.3.

Operator 3.17 `spatialDistribution`

```
spatialDistribution(
  in0 = in0, in1 = in1, x = x,
  positiveVelocity = ...,
  initialPoints = ...,
  initialValues = ...)
```

`spatialDistribution` allows approximation of variable-speed transport of properties. For further details, see section 3.7.4.2.

Operator 3.18 `getInstanceName`

```
getInstanceName()
```

Returns a string with the name of the model/block that is simulated, appended with the fully qualified name of the instance in which this function is called. For further details, see section 3.7.4.6.

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

3.7.4.1 delay

[`delay` *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, `delay` *could break algebraic loops. For simplicity, this is not supported because the minimum delay time has to be given 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.*]

3.7.4.2 spatialDistribution

[*Many applications involve the modelling of variable-speed transport of properties. One option to model this infinite-dimensional system is to approximate it by an ODE, but this requires a large number of state variables and might introduce either numerical diffusion or numerical oscillations. Another option is to use a built-in operator that keeps track of the spatial distribution of $z(x,t)$, by suitable sampling, interpolation, and shifting of the stored distribution. In this case, the internal state of the operator is hidden from the ODE solver.*]

`spatialDistribution` allows the infinite-dimensional problem below to be solved efficiently with good accuracy

$$\begin{aligned}\frac{\partial z(x,t)}{\partial t} + v(t)\frac{\partial z(x,t)}{\partial x} &= 0.0\\ z(0.0,t) &= \text{in}_0(t) \text{ if } v \geq 0\\ z(1.0,t) &= \text{in}_1(t) \text{ if } v < 0\end{aligned}$$

where $z(x,t)$ is the transported quantity, x is the normalized spatial coordinate ($0.0 \leq x \leq 1.0$), t is the time, $v(t) = \text{der}(x)$ is the normalized transport velocity and the boundary conditions are set at either $x = 0.0$ or $x = 1.0$, depending on the sign of the velocity. The calling syntax is:

```
(out0, out1) = spatialDistribution(in0, in1, x, positiveVelocity,
                                   initialPoints = {0.0, 1.0},
                                   initialValues = {0.0, 0.0});
```

where `in0`, `in1`, `out0`, `out1`, and `x` are all subtypes of `Real`, `positiveVelocity` is a `Boolean`, `initialPoints` and `initialValues` are arrays of subtypes of `Real` of equal size, containing the x coordinates and the z values of a finite set of points describing the initial distribution of $z(x, t0)$. The `out0` and `out1` are given by the solutions at $z(0.0,t)$ and $z(1.0,t)$; and `in0` and `in1` are the boundary conditions at $z(0.0,t)$ and $z(1.0,t)$ (at each point in time only one of `in0` and `in1` is used). Elements in the `initialPoints` array must be sorted in non-descending order. The operator can not be vectorized according to the vectorization rules described in section 12.4.6. The operator can be vectorized only with respect to the arguments `in0` and `in1` (which must have the same size), returning vectorized outputs `out0` and `out1` of the same size; the arguments `initialPoints` and `initialValues` are vectorized accordingly.

The solution, z, can be described in terms of characteristics:

$$z(x + \int_t^{t+\beta} v(\alpha)\text{d}\alpha, t+\beta) = z(x,t), \quad \text{for all } \beta \text{ as long as staying inside the domain}$$

This allows the direct computation of the solution based on interpolating the boundary conditions.

`spatialDistribution` can be described in terms of the pseudo-code given as a block:

```
block spatialDistribution
  input Real in0;
  input Real in1;
  input Real x;
  input Boolean positiveVelocity;
  parameter Real initialPoints(each min=0, each max=1)[:] = {0.0, 1.0};
  parameter Real initialValues[:] = {0.0, 0.0};
  output Real out0;
  output Real out1;
protected
  Real points[:];
```

```
  Real values[:];
  Real x0;
  Integer m;
algorithm
  /* The notation
   *   x <and then> y
   * is used below as a shorthand for
   *   if x then y else false
   * also known as "short-circuit evaluation of x and y".
   */
  if positiveVelocity then
    out1 := interpolate(points, values, 1 - (x - x0));
    out0 := values[1]; // Similar to in0 but avoiding algebraic loop.
  else
    out0 := interpolate(points, values, 0 - (x - x0));
    out1 := values[end]; // Similar to in1 but avoiding algebraic loop.
  end if;
  when <acceptedStep> then
    if x > x0 then
      m := size(points, 1);
      while m > 0 <and then> points[m] + (x - x0) >= 1 loop
        m := m - 1;
      end while;
      values := cat(1,
                    {in0},
                    values[1:m],
                    {interpolate(points, values, 1 - (x - x0))});
      points := cat(1, {0}, points[1:m] .+ (x-x0), {1});
    elseif x < x0 then
      m := 1;
      while m < size(points, 1) <and then> points[m] + (x - x0) <= 0 loop
        m := m + 1;
      end while;
      values := cat(1,
                    {interpolate(points, values, 0 - (x - x0))},
                    values[m:end],
                    {in1});
      points := cat(1, {0}, points[m:end] .+ (x - x0), {1});
    end if;
    x0 := x;
  end when;
initial algorithm
  x0 := x;
  points := initialPoints;
  values := initialValues;
end spatialDistribution;
```

[*Note that the implementation has an internal state and thus cannot be described as a function in Modelica;* `initialPoints` *and* `initialValues` *are declared as parameters to indicate that they are only used during initialization.*

The infinite-dimensional problem stated above can then be formulated in the following way:

```
der(x) = v;
(out0, out1) = spatialDistribution(in0, in1, x, v >= 0,
                                   initialPoints, initialValues);
```

Events are generated at the exact instants when the velocity changes sign – if this is not needed, `noEvent` *can be used to suppress event generation.*

If the velocity is known to be always positive, then `out0` *can be omitted, e.g.:*

```
der(x) = v;
(, out1) = spatialDistribution(in0, 0, x, true, initialPoints, initialValues);
```

Technically relevant use cases for the use of `spatialDistribution` *are modeling of electrical transmission lines, pipelines and pipeline networks for gas, water and district heating, sprinkler systems, impulse propagation in elongated bodies, conveyor belts, and hydraulic systems. Vectorization is needed for pipelines where more than one quantity is transported with velocity* `v` *in the example above.*]

3.7.4.3 cardinality (deprecated)

[`cardinality` *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.*

]

[`cardinality` *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;
```

]

The cardinality is counted after removing conditional components, and shall not be applied to expandable connectors, elements in expandable connectors, or to arrays of connectors (but can be applied to the scalar elements of array of connectors). `cardinality` should only be used in the condition of assert and **`if`**-statements that do not contain `connect` and similar operators, see section 16.8.1).

3.7.4.4 homotopy

[*During the initialization phase of a dynamic simulation problem, it often happens that large nonlinear systems of equations must be solved by means of an iterative solver. The convergence of such solvers critically depends on the choice of initial guesses for the unknown variables. The process can be made more robust by providing an alternative, simplified version of the model, such that convergence is possible even without accurate initial guess values, and then by continuously transforming the simplified model into the actual model. This transformation can be formulated using expressions of this kind:*

$$\lambda \cdot \texttt{actual} + (1 - \lambda) \cdot \texttt{simplified}$$

in the formulation of the system equations, and is usually called a homotopy transformation. If the simplified expression is chosen carefully, the solution of the problem changes continuously with λ*, so by taking small enough steps it is possible to eventually obtain the solution of the actual problem.*

The operator can be called with ordered arguments or preferably with named arguments for improved readability.

It is recommended to perform (conceptually) one homotopy iteration over the whole model, and not several homotopy iterations over the respective non-linear algebraic equation systems. The reason is that the following structure can be present:

```
w = f1(x) // has homotopy
0 = f2(der(x), x, z, w)
```

Here, a non-linear equation system f_2 *is present.* `homotopy` *is, however used on a variable that is an "input" to the non-linear algebraic equation system, and modifies the characteristics of the non-linear*

algebraic equation system. The only useful way is to perform the homotopy iteration over f_1 and f_2 together.

The suggested approach is "conceptual", because more efficient implementations are possible, e.g., by determining the smallest iteration loop, that contains the equations of the first BLT block in which homotopy *is present and all equations up to the last BLT block that describes a non-linear algebraic equation system.*

A trivial implementation of homotopy *is obtained by defining the following function in the global scope:*

```
function homotopy
  input Real actual;
  input Real simplified;
  output Real y;
algorithm
  y := actual;
  annotation(Inline = true);
end homotopy;
```

]

[*Example 1: In electrical systems it is often difficult to solve non-linear algebraic equations if switches are part of the algebraic loop. An idealized diode model might be implemented in the following way, by starting with a "flat" diode characteristic and then move with* homotopy *to the desired "steep" characteristic:*

```
model IdealDiode
  ...
  parameter Real Goff = 1e-5;
protected
  Real Goff_flat = max(0.01, Goff);
  Real Goff2;
equation
  off = s < 0;
  Goff2 = homotopy(actual = Goff, simplified = Goff_flat);
  u = s * (if off then 1 else Ron2) + Vknee;
  i = s * (if off then Goff2 else 1 ) + Goff2*Vknee;
  ...
end IdealDiode;
```

]

[*Example 2: In electrical systems it is often useful that all voltage sources start with zero voltage and all current sources with zero current, since steady state initialization with zero sources can be easily obtained. A typical voltage source would then be defined as:*

```
model ConstantVoltageSource
  extends Modelica.Electrical.Analog.Interfaces.OnePort;
  parameter Modelica.Units.SI.Voltage V;
equation
  v = homotopy(actual = V, simplified = 0.0);
end ConstantVoltageSource;
```

]

[*Example 3: In fluid system modelling, the pressure/flowrate relationships are highly nonlinear due to the quadratic terms and due to the dependency on fluid properties. A simplified linear model, tuned on the nominal operating point, can be used to make the overall model less nonlinear and thus easier to solve without accurate start values. Named arguments are used here in order to further improve the readability.*

```
model PressureLoss
  import Modelica.Units.SI;
  ...
  parameter SI.MassFlowRate m_flow_nominal "Nominal mass flow rate";
  parameter SI.Pressure dp_nominal "Nominal pressure drop";
  SI.Density rho "Upstream density";
  SI.DynamicViscosity lambda "Upstream viscosity";
```

```
equation
  ...
  m_flow = homotopy(actual = turbulentFlow_dp(dp, rho, lambda),
                    simplified = dp/dp_nominal*m_flow_nominal);
  ...
end PressureLoss;
```

]

[*Example 4:* *Note that* homotopy shall not *be used to combine unrelated expressions, since this can generate singular systems from combining two well-defined systems.*

```
model DoNotUse
  Real x;
  parameter Real x0 = 0;
equation
  der(x) = 1-x;
initial equation
  0 = homotopy(der(x), x - x0);
end DoNotUse;
```

The initial equation is expanded into

$$0 = \lambda * \mathrm{der}(x) + (1 - \lambda)(x - x_0)$$

and you can solve the two equations to give

$$x = \frac{\lambda + (\lambda - 1)x_0}{2\lambda - 1}$$

which has the correct value of x_0 *at* $\lambda = 0$ *and of 1 at* $\lambda = 1$*, but unfortunately has a singularity at* $\lambda = 0.5$.]

3.7.4.5 semiLinear

(See definition of semiLinear in section 3.7.4). In some situations, equations with semiLinear 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:

- 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);
  ```

- 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.*]

3.7.4.6 getInstanceName

Returns a string with the name of the model/block that is simulated, appended with the fully qualified name of the instance in which this function is called.

[*Example:*

```
package MyLib
  model Vehicle
    Engine engine;
    ...
  end Vehicle;
  model Engine
    Controller controller;
    ...
  end Engine;
  model Controller
  equation
    Modelica.Utilities.Streams.print("Info from: " + getInstanceName());
  end Controller;
end MyLib;
```

If `MyLib.Vehicle` *is simulated, the call of* `getInstanceName()` *returns* `"Vehicle.engine.controller"`.]

If this function is not called inside a model or block (e.g., the function is called in a function or in a constant of a package), the return value is not specified.

[*The simulation result should not depend on the return value of this function.*]

3.7.5 Event-Related Operators with Function Syntax

The operators listed below are event-related operators with function syntax. The operators `noEvent`, `pre`, `edge`, and `change`, are vectorizable according to section 12.4.6.

Expression	*Description*	*Details*
`initial()`	Predicate for the initialization phase	Operator 3.19
`terminal()`	Predicate for the end of a successful analysis	Operator 3.20
`noEvent(`*expr*`)`	Evaluate *expr* without triggering events	Operator 3.21
`smooth(`*p*`,` *expr*`)`	Treat *expr* as *p* times continuously differentiable	Operator 3.22
`sample(`*start*`,` *interval*`)`	Periodic triggering of events	Operator 3.23
`pre(`*y*`)`	Left limit $y(t^-)$ of variable $y(t)$	Operator 3.24
`edge(`*b*`)`	Expands into `(`*b* `and not pre(`*b*`))`	Operator 3.25
`change(`*v*`)`	Expands into `(`*v* `<> pre(`*v*`))`	Operator 3.26
`reinit(`*x*`,` *expr*`)`	Reinitialize *x* with *expr*	Operator 3.27

Operator 3.19 initial

`initial()`

Returns `true` during the initialization phase and `false` otherwise.

[*Hereby,* `initial()` *triggers a time event at the beginning of a simulation.*]

Operator 3.20 terminal

`terminal()`

Returns `true` at the end of a successful analysis.

[*Hereby,* `terminal()` *ensures an event at the end of successful simulation.*]

Operator 3.21 noEvent

`noEvent(`*expr*`)`

`Real` elementary relations within *expr* are taken literally, i.e., no state or time event is triggered. No zero crossing functions shall be used to monitor any of the normally event-generating subexpressions inside *expr*. Inside functions, `noEvent` only makes a difference in combination with the function annotation `GenerateEvents = true` (see annotation 12.7). See also operator 3.22 *smooth* and section 8.5.

Operator 3.22 smooth

`smooth(`*p*`,` *expr*`)`

If $p \geq 0$ `smooth(`*p*`,` *expr*`)` returns *expr* and states that *expr* is *p* times continuously differentiable, i.e., *expr* 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 argument *p* should be a scalar `Integer` parameter expression. The only allowed types for *expr* in `smooth` are: `Real` expressions, arrays of allowed expressions, and records containing only components of allowed expressions.

`smooth` 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;
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)); // Needs noEvent.
```

]

Operator 3.23 sample

`sample(`*start*`,` *interval*`)`

Returns `true` and triggers time events at time instants $start + i \cdot interval$ for $i = 0, 1 \ldots$, and is only true during the first event iteration at those times. At event iterations after the first one at each event and during continuous integration the operator always returns `false`. The starting time *start* and the sample interval *interval* must be parameter expressions and need to be a subtype of `Real` or `Integer`. The sample interval *interval* must be a positive number.

Operator 3.24 pre

`pre(`*y*`)`

Returns the *left limit* $y(t^-)$ of variable $y(t)$ at a time instant t. At an event instant, $y(t^-)$ is the value of y after the last event iteration at time instant t (see comment below). Any subscripts in the component expression y must be parameter expressions. `pre` can be applied if the following three conditions are fulfilled simultaneously: (a) variable y is either a subtype of a simple type or is a record component, (b) y is a discrete-time expression (c) the operator is *not* applied in a **function** class.

[*This can be applied to continuous-time variables in* **when***-clauses, see section 3.8.5 for the definition of discrete-time expression.*]

The first value of `pre(`y`)` is determined in the initialization phase.

A new event is triggered if there is at least for one variable `v` such that `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 once `pre(v) == v` for all `v` appearing inside `pre(...)`.

[*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.*]

Operator 3.25 edge

`edge(`b`)`

Expands into `(`b `and not pre(`b`))` for `Boolean` variable b. The same restrictions as for `pre` apply (e.g., not to be used in **function** classes).

Operator 3.26 change

`change(v)`

Expands into `(`v `<> pre(`v`))`. The same restrictions as for `pre` apply.

Operator 3.27 reinit

`reinit(`x`,` *expr*`)`

In the body of a **when**-clause, reinitializes x with *expr* at an event instant. x is a scalar or array `Real` variable that is implicitly defined to have `StateSelect.always`.

[*It is an error if the variable cannot be selected as a state.*]

expr needs to be type-compatible with x. `reinit` can only be applied once for the same variable – either as an individual variable or as part of an array of variables. It can only be applied in the body of a **when**-clause in an equation section. See also section 8.3.6.

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 4.5 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

While many invalid models can be rejected based on the declared variabilities of variables alone (without the concept of expression variability), the following rules both help enforcing compliance of computed solutions to declared variability, and impose additional restrictions that simplify reasoning and reporting of errors:

- For an assignment `v := expr` or binding equation `v = expr`, `v` must be declared to be at least as variable as `expr`.
- For multiple return assignment `(x1, ..., xn) := expr` (see section 11.2.1.1), all of `x1`, ..., `xn` must be declared to be at least as variable as `expr`.
- When determining whether an equation can contribute to solving for a variable `v` (for instance, when applying the perfect matching rule, see section 8.4), the equation can only be considered contributing if the resulting solution would be at most as variable as `v`.

[*Example: The (underdetermined) model* `Test` *below illustrates two kinds of consequences due to variability constraints. First, it contains variability errors for declaration equations and assignments. Second, it illustrates the impact of variability on the matching of equations to variables, which can lead to violation of the perfect matching rule. Details of how variabilities are determined are given in the sections below. The* discrete-valued equation variability rule *mentioned in the comments below refer to the rule in section 3.8.5 that requires both sides of the* `Boolean` *equation to be discrete-time.*

```
model Constants
  parameter Real p1 = 1;
  constant Real c1 = p1 + 2; // Error, not a 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.
  Real x;
  Boolean b1 = noEvent(x > 1); // Error, since b1 is a discrete-time variable
                               // and noEvent(x > 1) is not discrete-time.
  Boolean b2;
  Integer i1;
  Integer i2;
algorithm
  i1 := x; // Error, assignment to variable of lesser variability.
equation
  /* The equation below can be rejected for two reasons:
   * 1. Discrete-valued equation variability rule requires both sides to be
   *    discrete-time.
   * 2. It violates the perfect matching rule, as no variable can be solved
   *    with correct variability using this equation.
   */
  b2 = noEvent(x > 1); // Error, see above.
  i2 = x;              // No variability error, and can be matched to x.
end Test;
```

]

3.8.1 Function Variability

The variability of function calls needs to consider both the variability of arguments directly given in the function and the variability of the used default arguments, if any. This is especially a concern for functions given as a short class, see section 12.1.3. This has additional implications for redeclarations, see definition 6.8. The purity of the function, see section 12.3, does not influence the variability of the function call.

[*The restrictions for calling functions declared as* `impure` *serve a similar purpose as the variability restrictions, see section 12.3, and thus it is not necessary to consider purity in the definition of variability. Consider a function reading an external file and returning some value from that file. Different uses can have the file updated before the simulation (as a parameter-expression), or during the simulation (as a discrete-time expression). Thus it depends on the use case and the specific file, not the function itself,*

and it would even be possible to update the file in continuous time (as part of an algorithm) and still use the same function.]

3.8.2 Constant Expressions

Constant expressions are:

- `Real`, `Integer`, `Boolean`, `String`, and **`enumeration`** literals.
- Constant variables, see section 4.5.
- 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.
- Some function calls are constant expressions regardless of the arguments:
  - `ndims(A)`

3.8.3 Evaluable Expressions

Evaluable expressions are:

- Constant expressions.
- Evaluable parameter variables, see section 4.5.
- Input variables in functions behave as though they were evaluable expressions.
- Except for the special built-in operators **`initial`**, `terminal`, `der`, `edge`, `change`, `sample`, and `pre`, a function or operator with evaluable subexpressions is an evaluable expression.
- The sub-expression **`end`** used in `A[... end ...]` if `A` is a variable declared in a non-**`function`** class.
- Some function calls are evaluable expressions even if the arguments are not:
  - `cardinality(c)`, see restrictions for use in section 3.7.4.3.
  - `size(A)` (including `size(A, j)` where `j` is an evaluable expression) if `A` is variable declared in a non-function class.
  - `Connections.isRoot(A.R)`
  - `Connections.rooted(A.R)`

3.8.4 Parameter Expressions

Parameter expressions are:

- Evaluable expressions.
- Non-evaluable parameter variables, see section 4.5.
- 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.
- Some function calls are parameter expressions even if the arguments are not:
  - `size(A, j)` where `j` is a parameter expression, if `A` is variable declared in a non-function class.

3.8.5 Discrete-Time Expressions

Discrete-time expressions are:

- Parameter expressions.
- Discrete-time variables, see section 4.5.
- 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, **initial equation**, or **initial algorithm**.
- Expressions in a clocked discrete-time partition, see section 16.8.1.
- Unless inside noEvent: Ordered relations (>, <, >=, <=) and the event generating functions ceil, floor, div, and integer, if at least one argument is non-discrete time expression and subtype of Real.

  [*These will generate events, see section 8.5. Note that* rem *and* mod *generate events but are not discrete-time expressions. In other words, relations inside* noEvent*, such as* noEvent(x>1)*, are not discrete-time expressions.*]
- The functions pre, edge, and change result in discrete-time expressions.
- Expressions in functions behave as though they were discrete-time expressions.

Inside an **if**-expression, **if**-clause, **while**-statement or **for**-clause, that is controlled by a non-discrete-time (that is continuous-time, but not discrete-time) switching expression and not in the body of a **when**-clause, it is not legal to have assignments to discrete-time variables, equations between discrete-time expressions, or real elementary relations/functions that should generate events.

[*The restriction above is necessary in order to guarantee that all equations for discrete-time variable are discrete-time expressions, and to ensure that crossing functions do not become active between events.*]

For a scalar or array equation expr1 = expr2 where neither expression is of base type Real, both expressions must be discrete-time expressions. For a record equation, the rule applies recursively to each of the components of the record. This is called the *discrete-valued equation variability rule*.

[*For a scalar equation, the rule follows from the observation that a discrete-valued equation does not provide sufficient information to solve for a continuous-valued variable. Hence, and according to the perfect matching rule (see section 8.4), such an equation must be used to solve for a discrete-valued variable. By the interpretation of* (B.1c) *in appendix B, it follows that one of* expr1 *and* expr2 *must be the variable, and the other expression its solution. Since a discrete-valued variable is a discrete-time expression, it follows that its solution on the other side of the equation must have at most discrete-time variability. That is, both sides of the equation are discrete-time expressions.*

For example, this rule shows that (outside of a **when***-clause)* noEvent *cannot be applied to either side of a* Boolean*,* Integer*,* String*, or* **enumeration** *equation, as this would result in a non-discrete-time expression.*

For an array equation, note that each array can have only one element type, so if one element is Real*, then all other entries must also be* Real*, possibly making use of standard type coercion, section 10.6.13. Hence, if the base type is not* Real*, all elements of the array are discrete-valued, allowing the argument above for a scalar equation to be applied elementwise to the array equation. That is, all array elements on both sides of the array equation will have discrete-time variability, showing that also the entire arrays* expr1 *and* expr2 *are discrete-time expressions.*

For a record equation, the components of the record have independent types, and the equation is seen as a collection of equations for the individual components of the record. In order to support records with components of mixed variability, a record equation with sides given by either record variables or record constructors is conceptually split before variability is determined.]

[*Example: Discrete-valued equation variability rule applied to record equations. In the first of the equations below, having a record constructor on both sides of the equation, the equation is conceptually split, and variabilities of* time *and* true *are considered separately. In the second equation, the* makeR *function call – regardless of inlining – means that the equation cannot be conceptually split into individual components of the record. The variability of the* makeR *call is continuous-time due to the* time *argument, which also becomes the variability of the* b *member of the call.*

```
record R
  Real x;
  Boolean b;
end R;

function makeR "Function wrapper around record constructor"
  input Real xx;
```

```
  input Boolean bb;
  output R r = R(xx, bb);
  annotation(Inline = true); // Inlining doesn't help.
end makeR;

model Test
  R r1, r2;
equation
  r1 = R(time, true);      // OK: Discrete-time Boolean member.
  r2 = makeR(time, true); // Error: Continuous-time Boolean member.
end Test;
```

]

3.8.6 Continuous-Time and Non-Discrete-Time Expressions

All expressions are continuous-time expressions including constant, parameter and discrete-time expressions. The term *non-discrete-time expression* refers to expressions that are neither constant, parameter nor discrete-time expressions. For example, **time** is a continuous-time built-in variable (see section 4.5) and **time** + 1 is a non-discrete-time expression. Note that plain **time** may, depending on context, refer to the continuous-time variable or the non-discrete-time expression.

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.

A protected element, `P`, in classes and components shall not be accessed via dot notation (e.g., `A.P`, `a.P`, `a[1].P`, `a.b.P`, `.A.P`; but there is no restriction on using `P` or `P.x` for a protected element `P`). They shall not be modified or redeclared except for modifiers applied to protected elements in a base class modification (not inside any component or class) and the modifier on the declaration of the protected element.

[*Example:*

```
package A
  model B
  protected
    parameter Real x;
  end B;
protected
  model C end C;
public
  model D
    C c; // Legal use of protected class C from enclosing scope
    extends A.B(x=2); // Legal modifier for x in derived class
                      // also x.start=2 and x(start=2) are legal.
    Real y=x; // Legal use of x in derived class
  end D;
  model E
    A.B a(x=2);  // Illegal modifier, also x.start=2 and x(start=2) are illegal
    A.C c;       // Illegal use of protected class C
    model F=A.C; // Illegal use of protected class C
  end E;
end A;
```

]

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. 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.

[*Example:*

```
record R
  Real x;
end R;
model M // wrong Modelica model
  R R; // not correct, since component name and type specifier are identical
equation
  R.x = 0;
end M;
```

]

4.3 Declaration Order

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 12.4.3.*
- *Records that are used as arguments to external functions, section 12.9.1.3.*
- *Enumeration literal order within enumeration types, section 4.9.5.*

]

4.4 Component Declarations

Component declarations are described in this section.

A *component declaration* is 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.

A *component* or *variable* is an instance (object) generated by a component declaration. Special kinds of components are scalar, array, and attribute.

4.4.1 Syntax

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 | stream ]
  [ discrete | parameter | constant ] [ input | output ]
```

```
type-specifier :
  name

component-list :
  component-declaration { "," component-declaration }

component-declaration :
  declaration [ condition-attribute ] comment

condition-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 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.

A class defined with **`partial`** in the *`class-prefixes`* is called a *partial* class. Such a class is allowed to be incomplete, and cannot be instantiated in a simulation model; useful, e.g., as a base class. See section 4.6.1 regarding short class definition semantics of propagating **`partial`**.

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 scalar non-negative evaluable expressions of type `Integer`, a reference to a type (which must an enumeration type or `Boolean`, see section 4.9.5), or the colon operator denoting that the array dimension is left unspecified (see section 10.1). All variants can also be part of short class definitions.

[*Example: Variables with array dimensions:*

```
model ArrayVariants
  type T = Real[:];                          // Unspecified size for type
  parameter T x = ones(4);
  parameter T y[3] = ones(3, 4);
  parameter Real a[2] = ones(2);             // Specified using Integer
  parameter Real b[2, 0] = ones(2, 0);       // Size 0 is allowed
  parameter Real c[:] = ones(0);             // Unspecified size for variable
  parameter Integer n = 0;
  Real z[n*2] = cat(1, ones(n), zeros(n));// Parameter expressions are allowed
  Boolean notV[Boolean] = {true, false};  // Indexing with type
end ArrayVariants;
```

]

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. These are a subset of the binding equations, see section 8.1. The declaration equation is of the form **x = expression** defined by a component declaration, where **expression** must not have higher variability than the declared component **x** (see section 3.8). Unlike other equations, a declaration equation can be overridden (replaced or removed) by an element modification.

For declarations of vectors and matrices, declaration equations are associated with each element.

Only components of the specialized classes **type**, **record**, **operator record**, and **connector**, or components of classes inheriting from `ExternalObject` may have declaration equations. See also the corresponding rule for algorithms, section 11.2.1.2.

4.4.2.2 Prefix Rules

A *prefix* is property of an element of a class definition which can be present or not be present, e.g., `final`, **public**, `flow`.

Type prefixes (that is, `flow`, `stream`, `discrete`, `parameter`, `constant`, `input`, `output`) shall only be applied for type, record, operator record, and connector components – see also record specialized class, section 4.7. This is further restricted below; some of these combinations of type prefixes and specialized classes are not legal.

An exception is `input` for components whose type is of the specialized class **function** (these can only be used for function formal parameters and has special semantics, see section 12.4.2). In this case, the `input` prefix is not applied to the elements of the component, and the prefix is allowed even if the elements of the component have `input` or `output` prefix.

In addition, instances of classes extending from `ExternalObject` may have type prefixes `parameter` and `constant`, and in functions also type prefixes `input` and `output`, see section 12.9.7.

Variables declared with the `stream` type prefix shall be a subtype of `Real`, or a **record** component where all the primitive elements shall be a subtype of `Real`. The members of the record may not have the `stream` type prefix. This is further restricted in section 15.1.

Variables declared with the `input` type prefix must not also have the prefix `parameter` or `constant`.

The type prefix `flow` of a component that is not a primitive element (see definition 9.1), is also applied to the elements of the component (this is done after verifying that the type prefixes occurring on elements of the component are correct). Primitive elements with the `flow` type prefix shall be a subtype of `Real`, `Integer`, or an operator record defining an additive group, see section 9.2.

The type prefixes `input` and `output` of a structured component (except as described above) are also applied to the elements of the component (this is done after verifying that the type prefixes occurring on elements of the component are correct).

When any of the type prefixes `flow`, `input` and `output` are applied for a structured component, no element of the component may have any of these type prefixes, nor can they have `stream` prefix. The corresponding rules for the type prefixes `discrete`, `parameter` and `constant` are described in section 4.5.5 for structured components.

[*The prefixes* `flow`, `stream`, `input` *and* `output` *could be treated more uniformly above, and instead rely on other rules forbidding combinations. The type prefix* `stream` *can be applied to structured components, specifically records. The type prefix* `flow` *can be applied to structured components, see section 9.2. Note that there are no specific restrictions if an operator record component has the type prefix* `flow`*, since the members of an operator record cannot have any of the prefixes* `flow`, `stream`, `input` *or* `output`.]

[*Example:* `input` *can only be used, if none of the elements has a* `flow`, `stream`, `input` *or* `output` *type prefix.*]

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 4.8.
- 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 4.8.

  [*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 4.8 (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 Component Variability Prefixes

The prefixes `discrete`, `parameter`, `constant` of a component declaration are called *variability prefixes* and are the basis for defining in which situation the variable values of a component are initialized (see section 8.5 and section 8.6) and when they are changed during simulation. Further details on how the

prefixes relate to component variability, as well as rules applying to components the different variabilities, are given in section 4.5.

4.4.4 Acyclic Bindings of Constants and Parameters

For a constant or parameter **v** with declaration equation, the expression of the declaration equation in the flattended model must not depend on **v** itself, neither directly nor indirectly via other variables' declaration equations. To satisfy this condition, dependencies shall be removed as needed by applying simplifications based on values of constants (except with `Evaluate = false`) and all other *evaluable parameters* (section 4.5) that don't depend on **v**. It is not permitted to expand a non-scalar declaration equation into scalar equations to satisfy the condition.

That the value of an evaluable parameter is used for these simplifications does not mean that it has to be determined during translation, but if **v** is found to be an evaluable parameter, then a Modelica tool will be able to break all cycles involving **v** by making some (possibly none or all) of the other evaluable parameters determined during translation. Hence, evaluation of a constant or evaluable parameter can never require solving systems of equations; they can always be sorted so that they can be solved one at a time with the natural causality (i.e., the declaration equation is used to determine the value of the component to which it belongs).

[*Example: Direct and indirect cyclic dependency:*

```
/* All of the following are illegal: */
parameter Real r = 2 * sin(r); // Depends directly on r.
parameter Real p = 2 * q;      // Indirect dependency on p via q = sin(p).
parameter Real q = sin(p);     // Indirect dependency on q via p = 2 * q.
```

]

[*Example: While declaration equations must not be cyclical, the use of initial equations can still introduce valid cyclic dependencies between parameters:*

```
  parameter Real p = 2 * q; // This is the only declaration equation.
  parameter Real q(fixed = false);
initial equation
  q = sin(p); // OK, not a declaration equation.
```

]

[*Example: Breaking cyclic dependency.*

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

final ABCD a;
// Illegal cyclic dependency 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.

partial model PartialLumpedVolume
  parameter Boolean use_T_start = true "= true, use T_start, otherwise h_start"
    annotation(Dialog(tab = "Initialization"), Evaluate = true);
  parameter Medium.Temperature T_start=if use_T_start then system.T_start else
      Medium.temperature_phX(p_start,h_start,X_start)
    annotation(Dialog(tab = "Initialization", enable = use_T_start));
  parameter Medium.SpecificEnthalpy h_start=if use_T_start then
      Medium.specificEnthalpy_pTX(p_start, T_start, X_start) else Medium.
    h_default
    annotation(Dialog(tab = "Initialization", enable = not use_T_start));
end PartialLumpedVolume;
```

```
// Cycle for T_start and h_start, but still valid since cycle disappears
// when evaluating use_T_start

// The unexpanded bindings have illegal cycles for both x and y
// (even if they would disappear if bindings were expanded).
model HasCycles
  parameter Integer n = 10;
  final constant Real A[3, 3] = [0, 0, 0; 1, 0, 0; 2, 3, 0];
  parameter Real y[3] = A * y + ones(3);
  parameter Real x[n] = cat(1, {3.4}, x[1:(n-1)]);
end HasCycles;
```

]

4.4.5 Conditional Component Declaration

A component declaration can have a `condition-attribute`: **`if`** *expression*.

[*Example:*

```
  parameter Integer level(min=1)=1;
  Motor motor;
  Level1 component1(J=J) if level==1 "Conditional component";
  Level2 component2 if level==2 "Conditional component";
  Level3 component3(J=component1.J) if level<2 "Conditional component";
  // Illegal modifier on component3 since component1.J is conditional
  // Even if we can see that component1 always exist if component3 exist
equation
  connect(component1..., ...) "Connection to conditional component 1";
  connect(component2.n, motor.n) "Connection to conditional component 2";
  connect(component3.n, motor.n) "Connection to conditional component 3";
  component1.u=0; // Illegal
```

]

The *expression* must be a `Boolean` scalar expression, and must be an evaluable expression.

[*An evaluable expression is required since it shall be evaluated at compile time.*]

A redeclaration of a component shall not include a condition attribute; and the condition attribute is kept from the original declaration (see section 6.4).

If the `Boolean` expression is false the component (including its modifier) is removed from the flattened DAE, and connections to/from the component are removed. A component declared with a condition-attribute can only be modified and/or used in connections.

[*Adding the component and then removing it ensures that the component is valid.*

If a `connect`*-equation defines the connection of a non-conditional component* `c1` *with a conditional component* `c2` *and* `c2` *is de-activated, then* `c1` *must still be a declared element.*

There are annotations to handle the case where the connector should be connected when activated, see section 18.8.]

4.5 Component Variability

As briefly mentioned in section 4.4.3, the component variability prefixes are the basis for defining *component variability*. Combined with some other information about the components and analysis of expression variability (section 3.8), they define the component variabilities as follows:

- A variable `vc` declared with `constant` prefix does not change during simulation, with a value that is unaffected even by the initialization problem (i.e., determined during translation). This is called a *constant*, or *constant variable*. For further details, see 4.5.1.
- A variable `ep` is called an *evaluable parameter variable* if all of the following applies:

  - It is declared with the `parameter` prefix.
  - It has `fixed = true`.
  - It does not have annotation `Evaluate = false`.
  - The declaration equation – or `start`-attribute if no declaration equation is given (see section 8.6) – is given by an evaluable expression (section 3.8.3).

  It is also simply called an *evaluable parameter*. An evaluable parameter does not change during transient analysis, with a value either determined during translation (similar to having prefix `constant`, and is then called an *evaluated parameter*) or by the initialization problem (similar to a *non-evaluable parameter*, see item below). At which of these stages the value is determined is tool dependent. For further details, see 4.5.2.

- A variable `np` declared with the `parameter` prefix, is called a *non-evaluable parameter variable* unless it is an evaluable parameter. It is also simply called a *non-evaluable parameter*. It does not change during transient analysis, with a value determined by the initialization problem. For further details, see 4.5.2.
- A *discrete-time variable* `vd` is a variable that is discrete-valued (that is, not of `Real` type) or assigned in a **`when`**-clause. The `discrete` prefix may be used to clarify that a variable is discrete-time. During transient analysis the variable can only change its value at event instants (see section 8.5). For further details, see 4.5.3.
- A *continuous-time variable* is a `Real` variable without any prefix that is not assigned in a **`when`**-clause. The variable can change both continuously and discontinuously at any time. For further details, see 4.5.4.

The term *parameter variable* or just *parameter* refers to a variable that is either an evaluable or non-evaluable parameter variable.

The variability of expressions and restrictions on variability for declaration equations is given in section 3.8.

[*Note that discrete-time* expressions *include parameter expressions, whereas discrete-time* variables *do not include parameter variables. The reason can intuitively be explained as follows:*

- *When discussing variables we also want to consider them as left-hand-side variables in assignments, and thus a lower variability would be a problem.*
- *When discussing expressions we only consider them as right-hand-side expressions in those assignment, and thus a lower variability can automatically be included; and additionally we have sub-expressions where lower variability is not an issue.*

For `Real` *variables we can distinguish two subtly different categories: discrete-time and piecewise constant, where the discrete-time variables are a subset of all piecewise constant variables. The* `Real` *variables declared with the prefix* `discrete` *is a subset of the discrete-time* `Real` *variables. For a* `Real` *variable, being discrete-time is equivalent to being assigned in a* **`when`***-clause. A variable used as argument to* `pre` *outside a* **`when`***-clause must be discrete-time.*

```
model PiecewiseConstantReals
  discrete Real xd1 "Must be assigned in a when-clause, discrete-time";
  Real xd2 "Assigned in a when-clause (below) and thus discrete-time";
  Real xc3 "Not discrete-time, but piecewise constant";
  Real x4 "Piecewise constant, but changes between events";
equation
  when sample(1, 1) then
    xd1 = pre(xd1) + 1;
    xd2 = pre(xd2) + 1;
  end when;
  // It is legal to use pre for a discrete-time variable outside of when
  xc3 = xd1 + pre(xd2);
  // But pre(xc3) would not be legal
  x4 = if noEvent(cos(time) > 0.5) then 1.0 else -1.0;
end PiecewiseConstantReals;
```

Tools may optimize code to only compute and store discrete-time variables at events. Tools may extend that optimization to piece-wise constant variables that only change at events (in the example above `xc3`*). As shown above variables can be piecewise constant, but change at times that are not events (in the example above* `x4`*). It is not clear how a tool could detect and optimize the latter case.*

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`*-equation, a* `parameter` *prefix allows removing 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 translation and usually not changed after they have been given a value. It can be used to represent mathematical constants, e.g.:

```
final constant Real PI = 4 * atan(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`*.]*

4.5.1 Constants

Constant variables (defined in section 4.5) shall have an associated declaration equation with a constant expression, if the constant is directly in the simulation model, or used in the simulation model. The value of a constant can be modified after it has been given a value, unless the constant is declared `final` or modified with a `final` modifier. A constant without an associated declaration equation can be given one by using a modifier.

By the acyclic binding rule in section 4.4.4, it follows that the value of a constant (or evaluable parameter, see below) to be used in simplifications is possible to obtain by evaluation of an evaluable expression where values are available for all component subexpressions.

4.5.2 Parameters

Parameter variables are divided into evaluable parameter variables and non-evaluable parameter variables, both defined in section 4.5.

By the acyclic binding rule in section 4.4.4, it follows that a value for an evaluable parameter is possible to obtain during translation, compare section 4.5.1. Making use of that value during translation turns the evaluable parameter into an evaluated parameter, and it must be ensured that the parameter cannot be assigned a different value after translation, as this would invalidate the use of the original value during translation.

[*Example: A particularly demanding aspect of this evaluation is the potential presence of external functions. Hence, if it is known that a parameter won't be used by an evaluable expression, a user can make it clear that the external function is not meant to be evaluated during translation by using* `Evaluate = false`*:*

```
import length = Modelica.Utilities.Strings.length; // Pure external function
parameter Integer n = length("Hello");             // Evaluable parameter
parameter Integer p = length("Hello")
  annotation(Evaluate = false);                    // Non-evaluable parameter
parameter Boolean b = false;                       // Evaluable parameter

/* Fulfillment of acyclic binding rule might cause evaluation of n;
 * to break the cycle, a tool might evaluate either b, n, or both:
 */
parameter Real x = if b and n < 3 then 1 - x else 0;

/* Fulfillment of acyclic binding rule cannot cause evaluation of p;
 * to break the cycle, evaluation of b is the only option:
 */
parameter Real y = if b and p < 3 then 1 - y else 0;
```

]

[*For a parameter in a valid model, presence of* `Evaluate` *(annotation 18.1) makes it possible to tell immediately whether it is an evaluable or non-evaluable parameter, at least as long as the warning described in annotation 18.1 isn't triggered. To see this, note that* `Evaluate = false` *makes it a non-evaluable parameter by definition, and that* `Evaluate = true` *would trigger the warning if the parameter is non-evaluable.*]

[*With every non-evaluable parameter, there is at least one reason why it isn't an evaluable parameter. This information is useful to maintain in tools, as it allows generation of informative error messages when a violation of evaluable expression variability is detected. For example:*

```
  parameter Integer n =
    if b then 1 else 2;     // Non-evaluable parameter due to variability of b.
  parameter Boolean b(fixed = false);
                            // Non-evaluable parameter due to fixed = false.
  Real[n] x;                // Variability error: n must be evaluable.
initial equation
  b = n > 3;
```

Here, a good error message for the variability error can include the information that the reason for `n` *being a non-evaluable parameter is that it has a dependency on the non-evaluable parameter* `b`.]

[*Related to evaluable parameters, the term* structural parameter *is also used in the Modelica community. This term has no meaning defined by the specification, and the meaning may vary from one context to another. One common meaning, however, is that in the context of a given tool, a parameter is called* structural *if the tool has decided to evaluate it because it controls some variation of the equation structure that the tool is unable to leave undecided during translation. With this interpretation of* structural parameter*, it follows that such a structural parameter must also be an evaluable parameter, while there are typically many evaluable parameters that are not structural.*]

4.5.3 Discrete-Time Variables

A discrete-time variable (defined in section 4.5) has a vanishing time derivative between events. Note that this is not the same as saying that `der(vd) = 0` almost everywhere, as the derivative is not even defined at the events. It is not allowed to apply `der` to discrete-time variables.

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 shall 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.

[*The restriction that discrete-valued variables (of type* `Boolean`*, etc) cannot be declared with continuous-time variability is one of the foundations of the expression variability rules that will ensure that any discrete-valued expression has at most discrete-time variability, see section 3.8.*]

4.5.4 Continuous-Time Variables

A continuous-time variable (defined in section 4.5) **vn** may have a non-vanishing time derivative (provided `der(vn)` is allowed this can be expressed as `der(vn) <> 0`) and may also change its value discontinuously at any time during transient analysis (see section 8.5). It may also contain a combination of these effects. Regarding existence of `der(vn)`, see operator 3.10.

4.5.5 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
```

]

4.6 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. The description consists of a class definition, a 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.

The object generated by a class is called an *instance*. An instance contains zero or more components (i.e., instances), equations, algorithms, and local classes. An instance has a type (section 6.3).

[*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.

An *element* is part of a class definition, and is 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.

```
class-definition :
  [ encapsulated ] class-prefixes
  class-specifier

class-prefixes :
  [ partial ]
  ( class | model | [ operator ] record | block | [ expandable ] connector |
    type |
  package | [ ( pure | impure ) ] [ operator ] function | operator )

class-specifier :
  long-class-specifier | short-class-specifier | der-class-specifier

long-class-specifier :
  IDENT description-string composition end IDENT
  | extends IDENT [ class-modification ] description-string
    composition end IDENT

short-class-specifier :
  IDENT "=" base-prefix name [ array-subscripts ]
  [ class-modification ] comment
  | IDENT "=" enumeration "(" ( [enum-list] | ":" ) ")" comment

der-class-specifier :
  IDENT "=" der "(" name "," IDENT { "," IDENT } ")" comment

base-prefix :
  [ input | output ]

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-clause ] ";" ]
  [ annotation-clause ";" ]
```

4.6.1 Short Class Definitions

A *short class definition* is a class definition in the form

```
class IDENT1 = type-specifier class-modification;
```

Except that **type-specifier** (the base-class) may be replaceable, and that the short class definition does not introduce an additional lexical scope for modifiers, it is identical to the longer form

```
class IDENT1
  extends type-specifier class-modification;
end IDENT1;
```

An exception to the above is that if the short class definition is declared as **encapsulated**, then the type-specifier and modifiers follow the rules for encapsulated classes and cannot be looked up in the enclosing scope.

[*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;
  encapsulated model Load2=.Resistor(R=2); // Ok
  encapsulated model LoadR=.Resistor(R=R); // Illegal
  Load a,b,c;
  ConstantSource ...;
  ...
end A;
```

The type-specifiers `.Resistor` *rely on global name lookup (see 5.3.2), due to the encapsulated restriction.*]

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 (_); see also section 4.6.2. 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: The types of* `f1` *and* `f2` *are identical:*

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

]

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 prefix **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; see also section 4.6.2.

[*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);
```

]

4.6.2 Combining Base Classes and Other Elements

It is not legal to combine equations, algorithms, components, non-empty base classes (see below), or protected elements with an extends from an array class, a class with non-empty *base-prefix*, a *simple type* (Real, Boolean, Integer, String and enumeration types), or any class transitively extending from an array class, a class with non-empty *base-prefix*, or a simple type.

Definition 4.1. ***Empty class.*** A class without equations, algorithms, or components, and where any base-classes are themselves empty. □

[*An empty class may contain annotations, such as graphics, and can be used more freely as base-class than other classes.*]

[*Example:*

```
model Integrator
  input Real u;
  output Real y = x;
  Real x;
equation
  der(x) = u;
end Integrator;

model Integrators = Integrator[3]; // Legal

model IllegalModel
  extends Integrators;
  Real x; // Illegal combination of component and array class
end IllegalModel;

connector IllegalConnector
  extends Real;
  Real y; // Illegal combination of component and simple type
end IllegalConnector;
```

]

4.6.3 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.

[*Example: The following example demonstrates parameterization of a local class:*

```
model C1
  type Voltage = Real(nominal=1);
  Voltage v1, v2;
end C1;

model 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.*]

4.7 Specialized Classes

Specialized kinds of classes (earlier known as *restricted classes*) **record**, **type**, **model**, **block**, **package**, **function** and **connector** have the properties of a general class, apart from restrictions. Moreover, they have additional properties called *enhancements*. The definitions of the specialized classes are given below (additional restrictions on inheritance are in section 7.1.3):

- **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). The elements of a record shall not have prefixes `input`, `output`, **inner**, **outer**, `stream`, or `flow`. Enhanced with implicitly available record constructor function, see section 12.6. The components directly declared in a record may only be of specialized class record or type.
- **type** – May only be predefined types, enumerations, array of type, or classes extending from type.
- **model** – The normal modeling class in Modelica.
- **block** – Same as **model** with the restriction that each public connector component of a **block** must have prefixes `input` and/or `output` for all connector variables that are neither `parameter` nor `constant`.

  [*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.*]
- **function** – See section 12.2 for restrictions and enhancements of functions. Enhanced to allow the function to contain an external function interface.

  [*Non-***function** *specialized classes do not have this property.*]
- **connector** – 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).

  Enhanced to allow `connect` to components of connector classes. The elements of a connector shall not have prefixes **inner**, or **outer**. May only contain components of specialized class **connector**, **record** and **type**.
- **package** – May only contain declarations of classes and constants. Enhanced to allow **import** of elements of packages. (See also chapter 13 on packages.)
- **operator record** – Similar to **record**; but operator overloading is possible, and due to this the typing rules are different, see chapter 6. It is not legal to extend from an **operator record** (or **connector** inheriting from **operator record**), except if the new class is an **operator record**

or **connector** that is declared as a short class definition, whose modifier is either empty or only modify the default attributes for the component elements directly inside the **operator record**. An **operator record** can only extend from an **operator record**. It is not legal to extend from any of its enclosing scopes. (See chapter 14).

- **operator** – May only contain declarations of functions. May only be placed directly in an **operator record**. (See also chapter 14).
- **operator function** – Shorthand for an **operator** with exactly one function; same restriction as **function** class and in addition may only be placed directly in an **operator record**.

  [*A function declaration*

```
operator function foo ... end foo;
```

  is conceptually treated as

```
operator foo function foo1
  ...
end foo1; end foo;
```

  ]

Additionally only components which are of specialized classes **record**, **type**, **operator record**, and connector classes based on any of those can be used as component references in normal expressions and in the left hand side of assignments, subject to normal type compatibility rules. Additionally components of connectors may be arguments of `connect`-equations, and any component can be used as argument to the `ndims` and `size`-functions, or for accessing elements of that component (possibly in combination with array indexing).

[*Example: Use of* **operator**:

```
operator record Complex
  Real re;
  Real im;
  ...
  encapsulated operator function '*'
    import Complex;
    input Complex c1;
    input Complex c2;
    output Complex result;
  algorithm
     result := Complex(re=c1.re*c2.re - c1.im*c2.im,
                       im=c1.re*c2.im + c1.im*c2.re);
  end '*';
end Complex;
record MyComplex
  extends Complex; // Error; extending from enclosing scope.
  Real k;
end MyComplex;
operator record ComplexVoltage = Complex(re(unit="V"),im(unit="V")); // allowed
```

]

4.8 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. 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 4.2. ***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`**, or **`operator 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, operator record, 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 4.3. ***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 B).*]

- 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, due to the balancing restrictions for connectors, see section 9.3.1.*]

- 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.*]

- For over-determined connectors, section 9.4, each spanning tree without any root node adds the difference between the size of the over-determined type or record and the size of the output of `equalityConstraint`.

  [*By definition this term is zero in simulation models, but relevant for checking component models. There are no other changes in the variable and equation count for models – but a restriction on the size of the output of* `equalityConstraint`*, section 9.3.1.*]

□

[*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 4.4. ***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. □

[*Here,* legal values *must respect 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 4.5. ***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`*. Note that this also applies to components inherited from a partial base-class provided the current class is non-partial.*]

- A component declared with the **inner** or **outer** prefix shall not be of a class having top-level public connectors containing inputs.
- In a declaration of a component of a record, connector, or simple type, modifiers can be applied to any element, and these are also considered for the equation count.

  [*Example:*

  ```
  Flange support(phi=phi, tau=torque1+torque2) if use_support;
  ```

  If `use_support=true`*, there are two additional equations for* `support.phi` *and* `support.tau` *via the modifier.*]
- In a declarations of a component of a **model** or **block** class, modifiers shall only contain redeclarations of replaceable elements and binding equations. The binding equations in modifiers for components may in these cases only be for parameters, constants, inputs and variables having a default binding equation. For the latter case of variables having a default binding equation the modifier may not remove the binding equation using **break**, see section 7.2.7.
- Modifiers of base-classes (on extends and short class definitions) shall only contain redeclarations of replaceable elements and binding equations. The binding equations follow the corresponding rules above, as if they were applied to the inherited component.
- 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:

- 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:* 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:

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

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*der(u) = p.i` *would be replaced by the equation* `C*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:* 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 variables for* `c` (`c.p.i`, `c.n.i`).

Local equations:

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

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 singular model.]

[*Example:* Example 3:

```
import Modelica.Units.SI;

partial model BaseProperties "Interface of medium model"
  parameter Boolean preferredStates = 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 model `BaseProperties` *together with its use in derived classes and as component relies on a special design pattern defined below. The variables* `p`, `h`, `Xi` *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, which is used in this case, is to give textual equations for them (as below); using* `connect`*-equations for these connectors would be possible (and would work) but is not part of the design pattern.*

This partial model defines that `T`, `d`, `u` *can be computed from the medium model, provided* `p`, `h`, `Xi` *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`, `Xi`*. 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 preferredStates then StateSelect.prefer else StateSelect.default),
    T(stateSelect =
      if preferredStates 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+`nXi` *(*`p`, `h`, `Xi`*: variables inside connectors defined in* `BaseProperties` *and inherited in* `SimpleAir` *)*

resulting in $5 +$ `nXi` *unknowns. The local equation size is:*

- 3 *(equations defined in* `SimpleAir`*), plus*
- $2 +$ `nXi` *(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 Modelica.Units.SI;

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(preferredStates = 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:*

- $4 + 2 \cdot$ `nXi` *(inside the* `port` *connector), plus*
- $2 +$ `nXi` *(variables* `U`, `M` *and* `MXi`*), plus*
- $2 +$ `nXi` *(the input variables in the connectors of the* `medium` *model)*

resulting in $8 + 4 \cdot$ `nXi` *unknowns; the local equation size is*

- $6 + 3 \cdot$ `nXi` *from the equation section, plus*
- $2 +$ `nXi` *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:*

- $4 + 2 \cdot$ `nXi` *(inside the* `port` *connector), plus*
- $2 +$ `nXi` *(the input variables in the connectors of the* `medium` *model)*

resulting in $6 + 3 \cdot$ `nXi` *unknowns, while the local equation size is*

- $4 + 2 \cdot$ `nXi` *from the equation section, plus*
- $2 +$ `nXi` *flow variables in the* `port` *connector.*

Therefore, `FixedBoundary_pTX` *is a locally balanced model. The predefined boundary variables* `p` *and* `Xi` *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.*]

4.9 Predefined Types and Classes

The *attributes* of the predefined variable types (`Real`, `Integer`, `Boolean`, `String`) and **`enumeration`** types are described below with Modelica syntax although they are predefined. 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.

The ⟨*value*⟩ in the definitions of the predefined types represents the value of an expresion of that type. Unlike attributes, the ⟨*value*⟩ of a component cannot be referred to by name; both access and modification of the value is made directly on the component.

[*Example: Accessing and modifying a variable value, using* `Real` *as example of a predefined type:*

```
model M
  record R
    Real u;
    Real v;
  end R;
  Real x = sin(time);       // Value modification.
  Real y(unit = "kg") = x; // Access value of x, and modify value of y.
  R r(u = y);               // Value modification of r.u.
equation
  r.v + x * x = 0;          // Access values of r.v and x.
end M;
```

Note that 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`]

It is not possible to combine extends from the predefined types, enumeration types, or this `Clock` type with other components.

The names `Real`, `Integer`, `Boolean` and `String` have restrictions similar to keywords, see section 2.3.3.

[*Hence, it is possible to define a normal class called* `Clock` *in a package and extend from it.*]

[*It also follows that the only way to declare a subtype of, e.g.,* `Real` *is to use the* **`extends`** *mechanism.*]

The definitions use `RealType`, `IntegerType`, `BooleanType`, `StringType`, `EnumType` as mnemonics corresponding to machine representations. These are called the *primitive types*.

Definition 4.6. ***Fallback value.*** In situations where the `start`-attribute would apply if provided, but the attribute is not provided, the *fallback value* shall be used instead. Tools are recommended to give diagnostics when the fallback value is used. The fallback values for variables of the different predefined types are defined below. □

4.9.1 Real Type

The following is the predefined `Real` type:

```
type Real // Note: Defined with Modelica syntax although predefined
  RealType ⟨value⟩; // Not an attribute; only 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;             // Initial value
  parameter BooleanType fixed = true,  // default for parameter/constant;
                              = false; // default for other variables
  parameter RealType nominal;             // Nominal value
  parameter BooleanType unbounded = false; // For error control
  parameter StateSelect stateSelect = StateSelect.default;
equation
  assert(min <= ⟨value⟩ and ⟨value⟩ <= max, "Variable value out of limit");
end Real;
```

The following attributes shall be evaluable: `quantity`, `unit`, `displayUnit`, `fixed`, and `stateSelect`.

The `unit` and `displayUnit` attributes may be either the empty string or a string matching *`unit-expression`* in chapter 19. The meaning of the empty string depends on the context. For the input and output components of a function, the empty string allows different units to be used in different calls to the function. For a non-function component, the empty string allows the unit (or display unit) to be inferred by the tool.

[*That* `displayUnit` *is evaluable allows tools to verify that the default display unit is consistent with the* `unit`. *Unlike the* `unit`, `displayUnit` *is just a default, tools may allow using other compatible display units for a translated model.*]

The `nominal` attribute is meant to be used for scaling purposes and to define tolerances in relative terms, see section 4.9.6.

The fallback value is the closest value to 0.0 consistent with the `min` and `max` bounds.

[*For external functions in C89,* `RealType` *maps to* `double`. *In the mapping proposed in Annex F of the C99 standard,* `RealType`/`double` *matches the IEC 60559:1989 (ANSI/IEEE 754-1985)* `double` *format.*]

4.9.2 Integer Type

The following is the predefined `Integer` type:

```
type Integer // Note: Defined with Modelica syntax although predefined
  IntegerType ⟨value⟩; // Not an attribute; only accessed without dot-notation
  parameter StringType quantity = "";
  parameter IntegerType min = -Inf, max = +Inf;
  parameter IntegerType start;          // Initial value
  parameter BooleanType fixed = true,  // default for parameter/constant;
                              = false; // default for other variables
equation
```

```
  assert(min <= ⟨value⟩ and ⟨value⟩ <= max, "Variable value out of limit");
end Integer;
```

The following attributes shall be evaluable: **quantity**, and **fixed**.

The minimal recommended number range for **IntegerType** is from -2147483648 to +2147483647, corresponding to a two's-complement 32-bit integer implementation.

The fallback value is the closest value to 0 consistent with the **min** and **max** bounds.

4.9.3 Boolean Type

The following is the predefined Boolean type:

```
type Boolean // Note: Defined with Modelica syntax although predefined
  BooleanType ⟨value⟩; // Not an attribute; only accessed without dot-notation
  parameter StringType quantity = "";
  parameter BooleanType start;          // Initial value
  parameter BooleanType fixed = true,   // default for parameter/constant;
                              = false, // default for other variables
end Boolean;
```

The following attributes shall be evaluable: **quantity**, and **fixed**.

The fallback value is false.

4.9.4 String Type

The following is the predefined String type:

```
type String // Note: Defined with Modelica syntax although predefined
  StringType ⟨value⟩; // Not an attribute; only accessed without dot-notation
  parameter StringType quantity = "";
  parameter StringType start;           // Initial value
  parameter BooleanType fixed = true,   // default for parameter/constant;
                              = false, // default for other variables
end String;
```

The following attributes shall be evaluable: **quantity**, and **fixed**.

A **StringType** value (such as ⟨*value*⟩ or other textual attributes of built-in types) may contain any Unicode data (and nothing else).

The fallback value is "".

4.9.5 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. 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 4.9.5.1. 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; see section 11.2.2.2. An element of an enumeration type can be accessed in an expression.

[*Uses of elements of enumeration type in expressions include indexing into an array.*]

[*Example:*

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

Setting attributes:

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

Using enumeration types as expressions:

```
Real x[DigitalCurrentChoices];

// Example using the type name to represent the range

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

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

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

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;
```

]

4.9.5.1 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⟩; // Not an attribute; only accessed without dot-notation
  parameter StringType quantity = "";
  parameter EnumType min = e1, max = en;
  parameter EnumType start;              // Initial value
  parameter BooleanType fixed = true,  // default for parameter/constant;
                              = false; // default for other variables
  constant EnumType e1 = ...;
  ...
  constant EnumType en = ...;
equation
  assert(min <= ⟨value⟩ and ⟨value⟩ <= max, "Variable value out of limit");
end E;
```

The following attributes shall be evaluable: `quantity`, and `fixed`.

The fallback value is the `min` bound.

[*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.*]

4.9.5.2 Conversion of Enumeration to String or Integer

The type conversion function `Integer(`⟨*expression of enumeration type*⟩`)` returns the ordinal number of the enumeration value `E.enumvalue`, to which the expression is evaluated, where `Integer(E.e1)` = 1, `Integer(E.en)` = n, 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.

4.9.5.3 Conversion of Integer to Enumeration

Whenever an enumeration type is defined, a type conversion function with the same name and in the same scope as the enumeration type is implicitly defined. This function can be used in an expression to convert an integer value to the corresponding (as described in section 4.9.5.2) enumeration value.

For an enumeration type named `EnumTypeName`, the expression `EnumTypeName(`⟨*Integer expression*⟩`)` returns the enumeration value `EnumTypeName.e` such that `Integer(EnumTypeName.e)` is equal to the original integer expression.

Attempting to convert an integer argument that does not correspond to a value of the enumeration type is an error.

[*Example:*

```
type Colors = enumeration ( RED, GREEN, BLUE, CYAN, MAGENTA, YELLOW );
```

Converting from `Integer` *to* `Colors`*:*

```
c = Colors(i);
c = Colors(10); // An error
```

]

4.9.5.4 Unspecified Enumeration

An enumeration type defined using `enumeration(:)` is unspecified and can be used as a replaceable enumeration type that can be freely redeclared to any enumeration type. There can be no enumeration variables declared using `enumeration(:)` in a simulation model.

4.9.6 Attributes start, fixed, nominal, and unbounded

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 lack of default allows the tool to propagate the nominal attribute based on equations, and if there is no value to propagate the tool should use a non-zero value, it may use additional information (e.g., `min` attribute) to find a suitable value, and as last resort use 1. If `unbounded = true` it indicates that the state may grow without bound, and the error in absolute terms shall be controlled.

[*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 tolerance for an integrator could be computed as* `tol * (abs(nominal) + (if x.unbounded then 0 else abs(x)))`. *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).*]

4.9.7 Other Predefined Types

4.9.7.1 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."
);
```

4.9.7.2 ExternalObject

See section 12.9.7 for information about the predefined type `ExternalObject`.

4.9.7.3 AssertionLevel

The predefined `AssertionLevel` enumeration type is used together with `assert`, section 8.3.7.

```
type AssertionLevel = enumeration(warning, error);
```

4.9.7.4 Connections

The package `Connections` is used for over-constrained connection graphs, section 8.3.9.

4.9.7.5 Graphical Annotation Types

A number of "predefined" record types and enumeration types for graphical annotations are described in chapter 18. These types are not predefined in the usual sense since they cannot be referenced in ordinary Modelica code, only within annotations.

4.9.7.6 Clock Types

See section 16.2.1 and section 16.3.

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 13.4. 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 and enumeration type conversion functions are ignored during type name lookup. Names of record classes and enumeration types are ignored during function name lookup.

[*The reason to ignore the implicitly defined names is that a record and the implicitly created record constructor function, see section 12.6, and an enumeration type and the implicitly created conversion function (section 4.9.5.3), have the same name.*]

5.3.1 Simple Name Lookup

A class declared with the keyword **encapsulated** (see *class-definition* in the grammar) is called an *encapsulated* class. By restricting name lookup inside a restricted class in ways defined in this chapter, the meaning of the class is made independent of where it is placed in a package hierarchy.

When an element, equation, or section is flattened, any simple name (not composed using dot notation) is first looked up sequentially among iteration variables (if any; see below), and then looked up sequentially in each member of the ordered set *of instance scopes (see section 5.6.1.1) corresponding to lexically 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.

The iteration variables are the implicitly declared iteration variable(s) if inside the body of a **for**-loop, section 8.3.2 and section 11.2.2, or the body of a reduction expression, section 10.3.4.

Reference to variables successfully looked up in an enclosing class is only allowed for variables declared as `constant`. The values of modifiers are thus resolved in the *instance* scope of which the modifier appears; if the use is in a modifier on a short class definition, see section 4.6.1.

This lookup in each *instance* 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**-clauses in the *instance* 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**-clauses in the *instance* scope. It is an error if this step produces matches from several unqualified imports.

The **import**-clauses defined in inherited classes are ignored for the lookup, i.e., **import**-clauses 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 not found, and if the first identifier denotes a component and the composite name is used as a function call, the lookup is also performed among the declared elements of the component, and must find a non-operator function. Each leading element, including the first one, must in this case be a scalar component, or `component[j]` where `component` is an array of components and the indices `j` are evaluable expressions and `component[j]` is a scalar. All identifiers of the rest of the name (e.g., `B` and `B.C`) must be classes. That is, the composite name is comprised of one or more component names (optionally with indexing), followed by one or more class names.
- If the identifier denotes a class, that class is temporarily flattened (as if instantiating a component without modifiers of this class, 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. The lookup will only find the element (assuming it exists) in the following cases:
  - If the class is declared as **package** or **operator** (but not **operator record** or **operator function**) all elements can be found.
  - An element can be found if it is declared as **encapsulated**.

– A deprecated case is that if the class satisfies the requirements for a **package** (without being declared as such), it is still treated as a **package**.

The class we look inside shall 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.*]

[*Example: Components and classes are part of the same name-space and thus a component cannot have the same name as its class or the first part of the class-name as that would prevent lookup of the class name.*

```
model A
  M M;     // Illegal, component 'M' prevents finding class 'M'
  P.Q P;   // Illegal, component 'P' prevents finding package 'P'
  .R R;    // Legal, see next section
  S.Q Q;   // Legal

  Y a;     // Illegal, component 'Y' (below) prevents finding class 'Y'
  Y.X b;   // Illegal, component 'Y' (below) prevents finding package 'Y'
  .Y c;    // Legal, see next section
  Real Y;
end A;
```

]

[*Note that an* **operator** *class may only contain declarations of functions and thus fulfills the requirements for a package (see section 4.7). In practice, the non-deprecated rules imply that we can call* `Complex.'-'.negate` *and* `Complex.'+'` *for the example in section 14.7. This requires that* **operator** `'-'` *and* **operator function** `'+'` *are declared as* **encapsulated** *as in the example.*]

5.3.3 Global Name Lookup

For a name starting with dot, e.g., `.A` (or `.A.B`, `.A.B.C` etc.) lookup is performed as follows:

- The first identifier (`A`) is looked up in the global scope. This is possible even if the class is encapsulated and **import**-clauses are not used for this. If there does not exist a class `A` in global scope this is an error.
- If the name is simple then the class `A` is the result of lookup.
- If the name is a composite name then the class `A` 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 shall not be partial.

[*The package-restriction ensures that global name lookup of component references can only find global constants.*]

5.3.4 Lookup of Imported Names

See section 13.2.1.

5.4 Inner Declarations - Instance Hierarchy Name Lookup

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.

Outer component declarations shall not have modifications (including binding equations). Outer class declarations should be defined using short-class definitions without modifications. However, see also section 5.5.

If the outer component declaration is a disabled conditional component (section 4.4.5) it is also ignored for the automatic creation of inner component (neither causing it; nor influencing the type of it).

An **outer** element reference in a simulation model requires that one corresponding **inner** element declaration exists or can be created in a unique way:

- If there are two (or more) **outer** declarations with the same name, both lacking matching **inner** declarations, and the **outer** declarations are not of the same class it is an error.
- If there is one (or more) **outer** declarations of a partial class it is an error.
- In other cases, i.e., if a unique non-partial class is used for all **outer** declarations of the same name lacking a matching inner declaration, then an **inner** declaration of that class is automatically added at the top of the model and a diagnostic is given.
- The annotations defined in section 18.8 does not affect this process, other than that:
  - `missingInnerMessage` can be used for the diagnostic (and possibly error messages)

An **outer** element component may be of a partial class (but the referenced **inner** component must be of a non-partial class).

[**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* **inner** *declaration in a more deeply nested level of the model hierarchy.*]

[*Example: Simple Example:*

```
class A
  outer Real T0;
  ...
end A;
class B
  inner Real T0=1;
  A a1, a2; // B.T0, B.a1.T0 and B.a2.T0 will have the same value
  A a3(T0=4); // Illegal as T0 is an outer 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 Field Functions Using Inner/Outer

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

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

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

class D
  outer function fc = A;
  ...
equation
  y = fc(u);
end D;

class C
  inner function fc = B; // define function to be actually used
  D d; // The equation is now treated as y = B(u)
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 follows the above rules for **inner** and another that follows 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**.]

Modifications of elements declared with both the prefixes **inner** and **outer** may have modifications, 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 enables 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 two major steps:

1. Instantiation process
2. Generation of the flat equation system

The result is an equation system of all equations/algorithms, initial equations/algorithms and instances of referenced functions. Modifications of constants, parameters and variables are included in the form of equations.

The constants, parameters and variables are defined by globally unique identifiers and all references are resolved to the identifier of the referenced variable. No other transformations are performed.

5.6.1 Instantiation

The instantiation is performed in two steps. First a class tree is created and then from that an instance tree for a particular model is built up. This forms the basis for derivation of the flat equation system.

An implementation may delay and/or omit building parts of these trees, which means that the different steps can be interleaved. If an error occurs in a part of the tree that is not used for the model to be instantiated the corresponding diagnostics can be omitted (or be given). However, errors that should only be reported in a simulation model must be omitted there, since they are not part of the simulation model.

5.6.1.1 The Class Tree

All necessary libraries including the model which is to be instantiated are loaded (e.g., from a file system) and form a so called *class tree*. This tree represents the syntactic information from the class definitions. It contains also all modifications at their original locations in syntactic form. The builtin classes are put into the unnamed root of the class tree.

[*The class tree is built up directly during parsing of the Modelica texts. For each class a local tree is created which is then merged into the one big tree, according to the location of the class in the class hierarchy. This tree can be seen as the abstract syntax tree (AST) of the loaded libraries.*]

5.6.1.2 The Instance Tree

The output of the instantiation process is an *instance tree*. The instance tree consists of nodes representing the elements of a class definition from the class tree. For a component the subtree of a particular node is created using the information from the class of the `component-clause` and a new modification environment as result of merging the current modification environment with the modifications from the current element declaration (see section 7.2.3).

The instance tree has the following properties:

- It contains the instantiated elements of the class definitions, with redeclarations taken into account and merged modifications applied.
- Each instance knows its source class definition from the class tree and its modification environment.
- Each modification knows its instance scope.

The instance tree is used for lookup during instantiation. To be prepared for that, it has to be based on the structure of the class tree with respect to the class definitions. The builtin classes are instantiated and put in the unnamed root prior to the instantiation of the user classes, to be able to find them.

[*The existence of the two separate trees (instance tree and class tree) is conceptual. Whether they really exist or are merged into only one tree or the needed information is held completely differently is an implementation detail. It is also a matter of implementation to have only these classes instantiated which are needed to instantiate the class of interest.*]

A node in the instance tree is the instance scope for the modifiers and elements syntactically defined in the class it is instantiated from. The instance scope is the starting point for name lookup.

[*If the name is not found the lookup is continued in the instance scope corresponding to the lexically enclosing class.* **`extends`***-clauses are treated as unnamed nodes in the instance tree – when searching for an element in an instance scope the search also recursively examines the elements of the* **`extends`***-clauses. Except that inherited* **`import`***-clauses are ignored.*]

5.6.1.3 The Instantiation Procedure

The instantiation is a recursive procedure with the following inputs:

- the class to be instantiated (current class)
- the modification environment with all applicable redeclarations and merged modifications (initially empty)
- a reference to the node of the instance tree, which the new instance should go into (parent instance)

The instantiation starts with the class to be instantiated, an empty modification environment, and an unnamed root node as parent node.

During instantiation all lookup is performed using the instance tree, starting from the instance scope of the current element. References in modifications and equations are resolved later (during generation of flat equation system) using the same lookup.

5.6.1.4 Steps of Instantiation

The element itself A *partially instantiated* class or component is an element that is ready to be instantiated; a partially instantiated element (i.e., class or component) is comprised of a reference to the original element (from the class tree) and the modifiers for that element (including a possible redeclaration).

The possible redeclaration of the element itself takes effect.

The class of a partially instantiated component is found in the instance tree (using the redeclaration if any), modifiers merged to that class forming a new partially instantiated class that is instantiated as below.

The local contents of the element For local classes and components in the current class, instance nodes are created and inserted into the current instance. Modifiers (including class redeclarations) are merged and associated with the instance and the element is partially instantiated.

[*The partially instantiated elements are used later for lookup during the generation of the flat equation system and are instantiated fully, if necessary, using the stored modification environment.*]

Equations, algorithms, and annotations of the class and the component declaration are copied to the instance without merging.

[*The annotations can be relevant for simulations, e.g., annotations for symbolic processing (annotation 18.1), simulation experiments (annotation 18.2) or functions (section 12.7 and section 12.9).*]

The **extends**-clauses are not looked up, but empty **extends**-clause nodes are created and inserted into the current instance (to be able to preserve the declaration order of components).

The inherited contents of the element Classes of **extends**-clauses of the current class are looked up in the instance tree, modifiers (including redeclarations) are merged, the contents of these classes are partially instantiated using the new modification environment, and are inserted into an **extends**-clause node, which is an unnamed node in the current instance that only contains the inherited contents from that base class.

The classes of **extends**-clauses are looked up before and after handling **extends**-clauses; and it is an error if those lookups generate different results.

At the end, the current instance is checked whether their children (including children of **extends**-clauses) with the same name are identical and only the first one of them is kept. It is an error if they are not identical.

[*Only keeping the first among the children with the same name is important for function arguments where the order matters.*]

Recursive instantiation of components Components (local and inherited) are recursively instantiated.

[*Example: As an example, consider:*

```
model M
  model B
    A a;
    replaceable model A = C;
    type E = Boolean;
  end B;
  B b(redeclare model A = D (p=1));
  partial model C
```

```
    E e;
  end C;

  model D
    extends C;
    parameter E p;
    type E = Integer;
  end D;

  type E = Real;
end M;
```

To recursively instantiate `M` *allowing the generation of flat equation system we have the following steps (not including checks):*

1. *Instantiate* `M`*, which partially instantiates* `B`*,* `b`*,* `C`*,* `D`*, and* `E`*.*
2. *Instantiate* `M.b`*:*
   - 2.1. *First find the class* `B` *in* `M` *(the partially instantiated elements have correct name allowing lookup)*
   - 2.2. *Instantiate the partially instantiated* `M.B` *with the modifier* `redeclare model A=D(p=1)`*.*
   - 2.3. *Partially instantiate* `M.b.a` *(no modifier), and* `M.b.A` *(with modifier* `=D(p=1)`*).*
3. *Instantiate* `M.b.a`*:*
   - 3.1. *First find the class* `A` *in* `M.b` *(the partially instantiated elements have correct name allowing lookup).*
   - 3.2. *Instantiate the partially instantiated* `M.b.A` *with the modifier* `=D(p=1)`*.*
     - 3.2.1. *Find the base class* `=D` *from the modifier. This performs lookup for* `D` *in* `M`*, and finds the partially instantiated class* `D`*.*
     - 3.2.2. *Instantiate the base class* `M.D` *with modifier* `p=1`*, and insert as unnamed node in* `M.b.A`*.*
       - 3.2.2.1. *Partially instantiate the component* `p` *with modifier* `=1`*.*
       - 3.2.2.2. *Find the base class* `C` *in* `M.D`*. Since there is no local element called* `C` *the search is then continued in* `M` *and finds the partially instantiated class* `M.C`*.*
       - 3.2.2.3. *Instantiate the base class* `M.C` *as below.*
4. *Instantiate the base class* `M.C` *inserting the result into unnamed node in* `M.b.a`*:*
   - 4.1. *Partially instantiate* `e`*.*
   - 4.2. *Instantiate* `e` *which requires finding* `E`*. First looking for* `E` *in the un-named node for* `extends M.C`*, and, since there is no local element* `E` *the search is then continued in* `M` *(which lexically encloses* `M.C`*) and finds* `E` *class inheriting from* `Real`*. The* `e` *is then instantiated using class* `E` *inheriting from* `Real`*.*
5. *Instantiate* `M.b.a.p`*:*
   - 5.1. *First the class* `E` *in* `M.b.a` *finding* `E` *class inheriting from* `Integer`*.*
   - 5.2. *Instantiate the* `M.b.a.p` *using the class* `E` *inheriting from* `Integer` *with modifier* `=1`*.*
   - 5.3. *Instantiate the base class* `Integer` *with modifier* `=1`*, and insert as unnamed node in* `M.b.a.p`*.*

An implementation can use different heuristics to be more efficient by re-using instantiated elements as long as the resulting flat equation system is identical.

Note that if `D` *was consistently replaced by* `A` *in the example above the result would be identical (but harder to read due to two different classes called* `A`*).*]

5.6.2 Generation of the Flat Equation System

During this process, all references by name in conditional declarations, modifications, dimension definitions, annotations, equations and algorithms are resolved to the real instance to which they are referring to, and the names are replaced by the global unique identifier of the instance.

[*This identifier is normally constructed from the names of the instances along a path in the instance tree (and omitting the unnamed nodes of* **extends***-clauses), separated by dots. Either the referenced instance belongs to the model to be simulated the path starts at the model itself, or if not, it starts at the unnamed root of the instance tree, e.g., in case of a constant in a package.*]

[*To resolve the names, a name lookup using the instance tree is performed, starting at the instance scope (unless the name is fully qualified) of the modification, algorithm or equation. If it is not found locally the search is continued at the instance of the lexically enclosing class of the scope (this is normally not equal to the parent of the current instance), and then continued with their parents as described in section 5.3. If the found component is an outer declaration, the search is continued using the direct parents in the instance tree (see section 5.4). If the lookup has to look into a class which is not instantiated yet (or only partially instantiated), it is instantiated in place.*]

The flat equation system consists of a list of variables with dimensions, flattened equations and algorithms, and a list of called functions which are flattened separately. A flattened function consists of an algorithm or **external**-clause and top-level variables (variables directly declared in the function or one of its base classes) – which recursively can contain other variables; the list of non-top-level variables is not needed.

The instance tree is recursively walked through as follows for elements of the class (if necessary a partially instantiated component is first instantiated):

- At each visited component instance, the name is inserted into the variables list. Then the conditional declaration expression is evaluated if applicable.
  - The variable list is updated with the actual instance
  - The variability information and all other properties from the declaration are attached to this variable.
  - Dimension information from the declaration and all enclosing instances are resolved and attached to the variable to define their complete dimension.
  - If it is of record or simple type (`Boolean`, `Integer`, enumeration, `Real`, `String`, `Clock`, `ExternalObject`):
    - In the modifications of *value* attribute references are resolved using the instance scope of the modification. An equation is formed from a reference to the name of the instance and the resolved modification value of the instance, and included into the equation system. Except if the value for an element of a record is overridden by the value for an entire record; section 7.2.3.
  - If it is of simple type (`Boolean`, `Integer`, enumeration, `Real`, `String`, `Clock`, `ExternalObject`):
    - In the modifications of *non-value* attributes, e.g., **start**, **fixed** etc. references are resolved using the instance scope of the modification. An equation is formed from a reference to the name of the instance appended by a dot and the attribute name and the resolved modification value of the instance, and included into the equation system.
  - If it is of a non-simple type the instance is recursively handled.
- If there are equation or algorithm sections in the class definition of the instance, references are resolved using the instance scope of the instance and are included in the equation system. Some references – in particular to non simple, non record objects like connectors in `connect`-equations and states in `transition`-equations are not resolved yet and handled afterwards.
- Instances of local classes are ignored.
- The unnamed nodes corresponding to **extends**-clauses are recursively handled.

- If there are function calls encountered during this process, the call is filled up with default arguments as defined in section 12.4.1. These are built from the modifications of input arguments which are resolved using their instance scope. The called function itself is looked up in the instance tree. All used functions are flattened and put into the list of functions.
- Conditional components with false condition are removed afterwards and they are not part of the simulation model.

  [*Thus, e.g., parameters don't need values in them. However, type-error can be detected.*]
- Each reference is checked, whether it is a valid reference, e.g., the referenced object belongs to or is an instance, where all existing conditional declaration expressions evaluate to true or it is a constant in a package.

  [*Conditional components can be used in* `connect`*-equations, and if the component is conditionally disabled the* `connect`*-equation is removed.*]

This leads to a flattened equation system, except for `connect`- and `transition`-equations. These have to be transformed as described in chapter 9 and chapter 17. This may lead to further changes in the instance tree (e.g., from expandable connectors (section 9.1.3)) and additional equations in the flattened equation system (e.g., connection equations (section 9.2), generated equations for state machine semantics (section 17.3.4)).

[*After flattening, the resulting equation system is self contained and covers all information needed to transform it to a simulatable model, but the class and instance trees are still needed: in the transformation process, there might be the need to instantiate further functions, e.g., from* `derivative` *annotation or from* `inverse` *annotation etc., on demand.*]

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 (section 6.4 and section 6.5) between "interfaces" (section 6.2); this can also be viewed as sub-typing (section 6.2).

6.1 Interface Terminology

In this chapter, two kinds of terminology are 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.

Definition 6.1. ***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`.

[*E.g., a declaration* `Real x` *is part of the type (also called* interface*), but* **`import`** `A` *is not.*] □

Definition 6.2. ***Class type* or *inheritance interface*.** The "essential" part of the public *and protected* 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. □

Definition 6.3. ***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`.

[*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 may be present in* `B`.] □

If `A` is a subtype of `B`, then `B` is said to be a *supertype* of `A`.

Definition 6.4. ***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.

[*E.g., it is not allowed that these connectors have variables with the* `input` *prefix, because then they must be connected.*]

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. □

Definition 6.5. ***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.

[*E.g., an additional argument must have a default value.*] □

6.2 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 {`false`, `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 `A1` being a subtype of type `A` means that the set of values corresponding to type `A1` 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 4.9).

The concept of *interface* as defined in section 6.3 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 6.3). The *type* of the element is the set of values having the same interface, i.e., the same attributes.

A *subtype* `A1` in relation to another type `A`, means that the elements of the set corresponding to `A1` 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 `R2`*:* `record R2 Boolean b; Real x; Real y; end R2;`

An instance `r`*:* `R r;`

An instance `r2`*:* `R2 r2;`

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=false, x=1.2)`*,* `R(b=false, x=3.4)`*,* `R(b=true, x=1.2)`*,* `R(b=true, x=1.2, y=2)`*,* `R(b=true, x=1.2, a=2)`*, ...}. The statement that* `r` *has the type (or interface)* `R` *means that the value of* `r` *s to this infinite set.*

The type `R2` *is a subtype of* `R` *since its instances fulfill the additional property of having the component* `Real y;` *in all its values.*

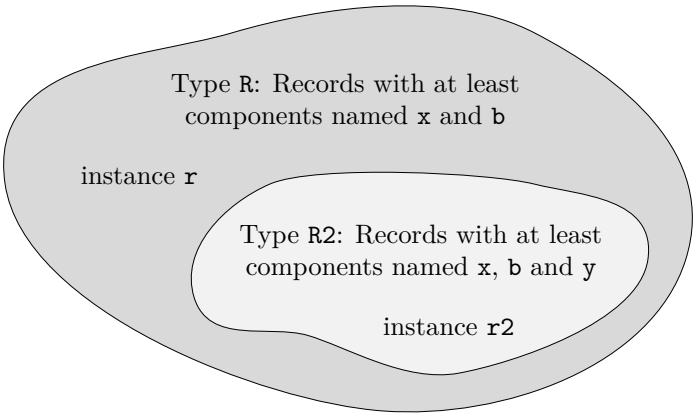


Figure 6.1: *The type* `R` *can be defined as the set of record values containing* `x` *and* `b`*. The subtype* `R2` *is the subset of values that all contain* `x`*,* `b`*, and* `y`*.*

]

6.3 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, and are different from *inheritance interface* and *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 6.3.1), and if not, the reference to the replaceable class it refers to.
- Whether it is a component or a class.
- Additional information about the element:
  - The `flow` or `stream` prefix.
  - Declared variability (`constant`, `parameter`, `discrete`).
  - The prefixes `input` and `output`.
  - The prefixes **`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`).
- Only for an **`operator record`** class and classes derived from `ExternalObject`: the full name of the operator record base class (i.e., the one containing the operations), or the derived class. See chapter 14 and section 12.9.7.

  The following item does not apply for an **`operator record`** class or class derived from `ExternalObject`, since the type is already uniquely defined by the full name.
- 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.*]

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.

[*The* constraining interface *consists of only the public elements, and if the declaration is replaceable the element is 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.*]

6.3.1 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*) all parts of the name satisfy the following:

- 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 is 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.*]

6.3.2 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 6.3.1), 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 about the element:
  - The `flow` or `stream` prefix.
  - Declared variability (`constant`, `parameter`, `discrete`).
  - The prefixes `input` and `output`.
  - The prefixes **`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**).

6.4 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 a subtype of the type of **B**, iff:

- **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 **A** has an **operator record** base class then **B** must also have one and it must be the same. If **A** does not have an operator record base class then **B** shall not have one. See chapter 14.
- If **A** is derived from **ExternalObject**, then **B** must also be derived from **ExternalObject** and have the same full name. If **A** is not derived from **ExternalObject** then **B** shall not be derived from **ExternalObject**. See section 12.9.7.
- If **B** is not replaceable then **A** shall not be replaceable.
- If **B** is transitively non-replaceable then **A** must be transitively non-replaceable (section 6.3.1). 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** shall 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** or **stream** prefix should be matched for compatibility.
  - Declared variability is ordered **constant** < **parameter** < **discrete** < continuous-time (**Real** without prefix), 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 prefix is as default inherited by the redeclaration (i.e., no need to repeat* **parameter** *when redeclaring a parameter).*]
  - The **input** and **output** prefixes 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** *prefix is inherited from the original declaration.*]
  - The **inner** and/or **outer** prefixes should be matched.

    [*For a redeclaration of an element the* **inner** *and/or* **outer** *prefixes are inherited from the original declaration (since it is not possible to have* **inner** *and/or* **outer** *as part of 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.
  - Conditional components are only compatible with conditional components. The conditions must have equivalent contents (similar to 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 an enumeration type defined as `(:)` then there is no restriction on the enumeration type `A`.
  - If `B` is a built-in type then `A` must also be of the same built-in type and vice versa.

[*Intuitively, that the type* `A` *is a subtype of the type of* `B` *means that all important elements of* `B` *are be present in* `A`.]

Plug-compatibility is a further restriction of compatibility (subtyping) defined in section 6.5, and further restricted for functions, see section 6.6. 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 6.5.

[*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.*]

6.5 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` *prefix 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 6.6. ***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 6.7. ***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, frition.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.]

6.6 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 6.8. ***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.
- If `A` is impure, then `B` must also be impure, compare section 12.3.

□

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.

Note that variability of function calls, see section 3.8.1, cannot be determined using just the interface of a function, as the variabilities of default argument expressions are not expressed by the interface. Hence a function redeclaration being function-compatible does not ensure that function calls will fulfill variability requirements, and tools must therefore check variability requirements separately.

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

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

function PointMassGravity
  extends GravityInterface;
  input Modelica.Units.SI.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;
  constant Real failed[:] = gravity({1, 0, 0}); // May fail
equation
  frame_a.f = gravity(frame_a.r0);
  // or gravity(position = frame_a.r0);
  frame_a.t = zeros(3);
end Body;

model PlanetSimulation
  parameter Modelica.Units.SI.Mass mSun = 2e30;
  function sunGravity = PointMassGravity(m = mSun);
  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`.

The constant `failed` *in* `planet1`*, will violate variability constraints, whereas it will work in* `planet2`*. The call* `gravity(frame_a.r0)` *will work in both of them.*]

6.7 Type Compatible Expressions

Certain expressions consist of an operator applied to two or more subexpressions (`A` and `B`). This includes:

- **`if`**-expressions, e.g., **`if`** `x` **`then`** `A` **`else`** `B`.
- Array expressions, e.g., `{A, B}`
- Binary operators if both operands are of simple types, e.g., `A + B`. Binary operators for other types are only defined for operator records, section 14.5, and do not necessarily require that the operands are type compatible with each other.

If the subexpressions satisfy the following restrictions they are called type compatible expressions. Otherwise the expression is illegal. The type of the full expression (e.g., **`if`** `x` **`then`** `A` **`else`** `B`) is also defined below.

- If `A` is a record expression, `B` must also be a record expression with the same named elements. In an expression that is not an array expression those elements must be type compatible. In an array expression the two records may contain elements with different sizes, but apart from that they must be type compatible. That generates a heterogenous array of records, see chapter 10. The type of the full expression is a record comprised of named elements that are type compatible with the corresponding named elements of both `A` and `B`.
- The rules for array expressions depend on the operation (the rules for binary operators are given in section 10.6 and for array concatenation in section 10.4.2). The rules for the remaining case of **`if`**-expressions and array-expressions are:
  - If `A` is an array expression then `B` must also be an array expression, and `ndims(A)` = `ndims(B)`. The type of the full expression is an array expression with elements compatible with the elements of both `A` and `B`. If both `size(A)` and `size(B)` are known and `size(A)` = `size(B)` then this defines the size of the full expression, otherwise the size of the full expression is not known until the expression is about to be evaluated. In case of an **`if`**-expression the size of the full expression is defined based on the branch selected, and for other cases `size(A)` = `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. The type of the full expression is `Real`, compare section 10.6.13, unless the operator is a relational operator (section 3.5) where the type of the full expression is `Boolean`.
- If `A` is an `Integer` expression then `B` must be a `Real` or `Integer` expression. For exponentiation and division the type of the full expression is `Real` (even if both `A` and `B` are `Integer`) see section 10.6.7 and section 10.6.5, for relational operators the type of the full expression is `Boolean`. In other cases the type of the full expression is `Real` or `Integer` (same as `B`), compare section 10.6.13.
- If `A` is a `Boolean` expression then `B` must be a `Boolean` expression and the type of the full expression is `Boolean`.
- If `A` is a `String` expression then `B` must be a `String` expression and the type of the full expression is `String`, unless the operator is a relational operator (section 3.5) where the type of the full expression is `Boolean`.
- If `A` is an enumeration expression then `B` must be an enumeration expression and the type of the full expression is enumeration expression, unless the operator is a relational operator (section 3.5) where the type of the full expression is `Boolean`. The enumeration expressions must be defined in terms of an enumeration type with the same enumeration literals in the same order.
- For array and **`if`**-expressions, if `A` has an **`operator record`** base class then `B` must also have an **`operator record`** base class, and it must be the same, and otherwise neither `A` nor `B` may have an **`operator record`** base class. This is also the **`operator record`** base class for the full expression, e.g., for **`if`** `(cond)` **`then`** `A` **`else`** `B`.
- If `A` is derived from `ExternalObject` then `B` must also be derived from `ExternalObject` and they must have the same full name; and otherwise neither `A` nor `B` may be derived from `ExternalObject`.

The common full name also defines the type of the full expression, e.g., for `if (cond) then A else B`.

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*. 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 derived class. In fact, the inherited contents is copied from the superclass into the derived class, 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 class `A` is called a *base class* of `B`, if `B` extends `A`. The converse relation is then expressed as `B` being a *derived class* of `A`, or as `B` being *derived from* `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, as explained below.

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

```
extends-clause :
   extends name [ class-or-inheritance-modification ] [ annotation-clause ]
```

The name of the base class is looked up in the partially flattened enclosing class (section 5.2) of the **`extends`**-clause. If the optional *`class-or-inheritance-modification`* contains any *`inheritance-modification`* the base class is then modified as described in section 7.4. The possibly modified 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
- a *`class-modification`* constructed from all `argument` of the *`inheritance-modification`*

in that order.

[*Example:*

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

class B
```

```
  extends A(b = 2);
end B;

class C
  extends B(a = 1);
end C;
```

]

The elements of the flattened base class become elements of the flattened enclosing class, and are added at the place of the **extends**-clause: specifically components and classes, the equation sections, algorithm sections, optional **external**-clause, and the contents of the annotation at the end of the class, 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, the first element in order (can be either inherited or local) is kept. It is recommended to give a warning for this case; unless it can be guaranteed that the identical contents will behave in the same way.

Otherwise the model is incorrect.

[*Clarifying order:*

```
function A
  input Real a;
  input Real b;
end A;

function B
  extends A;
  input Real a;
end B;
// The inputs of B are {a, b} in that order; the "input Real a;" is ignored.
```

]

Equations of the flattened base class that are syntactically equivalent to equations in the flattened enclosing class are discarded. This feature is deprecated, and it is recommended to give a warning when discarding them and for the future give a warning about all forms of equivalent equations due to inheritance.

[*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.

[*As stated in section 5.6.1.4, it is illegal for an* **extends***-clause to influence the lookup of the class name of any* **extends***-clause in the same class definition.*]

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 4.7, 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**-clause 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):

	Base Class											
Derived Class	**package**	**operator**	**function**	**operator function**	**type**	**record**	**operator record**	**expandable connector**	**connector**	**block**	**model**	**class**
package	yes											yes
operator		yes										yes
function			yes									yes
operator function			yes	yes								yes
type					yes							yes
record						yes						yes
operator record							yes					yes
expandable connector								yes				yes
connector					yes	yes	yes		yes			yes
block						yes				yes		yes
model						yes				yes	yes	yes
class												yes

If a derived class is inherited from another type of specialized class, then the result is a specialized class of the derived class type.

[*For example, if a* **block** *inherits from a* **record***, then the result is a* **block**.]

All specialized classes can be derived from **class**, provided that the resulting class fulfills the restriction of the specialized class. A **class** may only contain class definitions, annotations, and **extends**-clauses (having any other contents is deprecated).

[*It is recommended to use the most specific specialized class.*]

The specialized classes **package**, **operator**, **function**, **type**, **record**, **operator record**, and **expandable connector** can only be derived from their own kind and from **class**.

[*E.g., a package can only be base class for packages. All other kinds of classes can use the* **import***-clause to use the contents of a package.*]

[*Example:*

```
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 Require 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, see definition in section 6.3.1. For a replaceable component declaration without *constraining-clause* the class must use a class reference considered transitively non-replaceable.

[*The requirement to use a transitively non-replaceable name excludes the long form of redeclare, i.e.,* **redeclare model extends** `M` *... where* `M` *must be an inherited replaceable class.*]

[*The rule for a replaceable component declaration without* `constraining-clause` *implies that constraining classes are always transitively non-replaceable – both if explicitly given or implicitly by the declaration.*]

7.2 Modifications

A *modification* is part of an element. It modifies the instance generated by that element. A modification contains *element modifications* (e.g., `vcc(unit = "V") = 1000`) and *element-redeclarations* (e.g., **redeclare type** `Voltage = Real(unit="V")`).

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

- variable declarations
- short class definitions
- **extends**-clauses

A modifier modifies one or more declarations (definitions) from a class by changing some aspect(s) of the declarations (definitions). The most common kind of modifier just changes the *default value* or the `start`-attribute in a binding equation; the value and/or `start`-attribute should be compatible with the variable according to section 6.7.

An *element modification* overrides the declaration equation in the class used by the instance generated by the modified element.

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

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

]

A modification (e.g., `C1 c1(x = 5)`) is called a *modification equation*, if the modified variable (here: `c1.x`) is a non-parameter variable.

[*The modification equation is created, if the modified component (here:* `c1`*) is also created (see section 4.6). In most cases a modification equation for a non-parameter variable requires that the variable was declared with a declaration equation, see section 4.8; in those cases the declaration equation is replaced by the modification equation.*]

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 an *element-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). The modifier value (and class for redeclarations) is found in the context in which the modifier occurs, see also section 5.3.1.

[*Example: Scope for modifiers:*

```
model B
  parameter Real x;
  package Medium = Modelica.Media.PartialMedium;
end B;

model C
  parameter Real x = 2;
  package Medium = Modelica.Media.PartialMedium;
  B b(x = x, redeclare package Medium = Medium);
  // The 'x' and 'Medium' being modified are declared in the model B.
  // The modifiers '= x' and '= Medium' are found in the model C.
end C;

model D
  parameter Real x = 3;
  package Medium = Modelica.Media.PartialMedium;
  C c(b(x = x, redeclare package Medium = Medium));
  // The 'x' and 'Medium' being modified are declared in the model B.
  // The modifiers '= x' and '= Medium' are found in the model D.
end D;
```

]

When present, the description-string of a modifier overrides the existing description.

7.2.1 Syntax of Modifications and Redeclarations

The syntax is defined in the grammar, appendix A.2.5.

7.2.2 Modification Environment

The *modification environment* of a class contains arguments which modify elements of the class (e.g., parameter changes) when the class is flattened. The modification environment is built by merging class modifications, where outer modifications override inner modifications.

[*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 component overrides value modifiers for all components, and it is an error if this overrides a **final** prefix for a component, or if value for a simple component would override part of the value of a non-simple component. When merging modifiers each modification keeps its own **each** prefix.

[*Example: 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 following example demonstrates overriding part of non-simple component:

```
record A
  parameter Real x;
  parameter Real y;
end A;

model B
  parameter A a = A(2, 3);
end B;

model C
  B b1(a(x = 4)); // Error since attempting to override value for a.x when a
    has a value.
end C;
```

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;
  parameter Real 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*
x1	*none*
x2	*22*
x3.a	*33*
x4.b	*4*
x4.c	*44*
x5.a	x3.a
a	*55*
b	*66*
c	*77*

]

7.2.4 Single Modification

Two arguments of a modification shall not modify the same element, attribute, or description-string. When using qualified names the different qualified names starting with the same identifier are merged into one modifier. If a modifier with a qualified name has the **each** or **final** prefix, that prefix is only seen as applied to the final part of the name.

[*Example:*

```
class C1
  Real x[3];
end C1;
class C2 = C1(x = ones(3), x = ones(3)); // Error: x designated twice
class C3
  class C4
    Real x;
  end C4;
  C4 a(final x.unit = "V", x.displayUnit = "mV", x = 5.0);
  // Ok, different attributes designated (unit, displayUnit and value)
  // identical to:
  C4 b(x(final unit = "V", displayUnit = "mV") = 5.0));
end C3;
```

The following examples are incorrect:

```
m1(r = 1.5, r = 1.6) // Multiple modifier for r (its value)
m1(r = 1.5, r = 1.5) // Multiple modifier for r (its value) - even if identical
m1(r.start = 2, r(start = 3)) // Multiple modifier for r.start
m1(x.r = 1.5 "x", x.r(start = 2.0) "y")) // Multiple description-string for x.r
m1(r = R(), r(y = 2)) // Multiple modifier for r.y - both direct value and
                      // part of record
```

The following examples are correct:

```
m1(r = 1.5, r(start = 2.0))
m1(r = 1.6, r "x")
m1(r = R(), r(y(min = 2)))
```

]

7.2.5 Modifiers for Array Elements

The following rules apply to modifiers:

- 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 enclosing array (with regard to the position of **each**). In case of nested modifiers this implies it is applied individually to each element of each element of the enclosing array; see example. If the modified element is a vector and the modifier does not contain the **each** prefix, 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, until the last element of the vector is applied to the last element of the array; it is an error if these

sizes do not match. Matrices and general arrays of elements are treated by viewing those as vectors of vectors etc.

- 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});
  parameter Real b = 0;
end B;
```

This implies `c[i].a[j] = j` *and* `c[i].d = i`.

```
model D
  B b(each c.a = {3, 4, 5}, c.d = {2, 3, 4, 5, 6});
  // Equivalent to:
  B b2(c(each a = {3, 4, 5}, d = {2, 3, 4, 5, 6}));
end D;
```

This implies `b.c[i].a[j] = 2+j` *and* `b.c[i].d = 1+i`.

```
model E
  B b[2](each c(each a = {1, 2, 3}, d = {1, 2, 3, 4, 5}), p = {1, 2});
  // Without the first each one would have to use:
  B b2[2](c(each a = {1, 2, 3}, d = fill({1, 2, 3, 4, 5}, 2)), p = {1, 2});
end E;
```

This implies `b[k].c[i].a[j] = j`, `b[k].c[i].d = i`, *and* `b[k].p = k`. *For* `c.a` *the additional (outer)* each *has no effect, but it is necessary for* `c.d`.

Specifying array dimensions after the type works the same as specifying them after the variable name.

```
model F
  Real fail1[2](each start = {1, 2}); // Illegal
  Real work1[2](each start = 1);      // Legal
  Real[2] fail2(each start = {1, 2}); // Illegal
  Real[2] work2(each start = 2);      // Legal
end F;
```

]

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.*]

[*Example: Final component modification.*

```
type Angle =
  Real(final quantity = "Angle", final unit = "rad", displayUnit = "deg");

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, will indirectly change tf.b to 2 * {3, 1}
  PI c2(tf(b = {1}));  // error, b is declared as final
end Test;
```

]

[*Example: Final class declaration.*

```
model Test2
  final model MyTF = TransferFunction(b = {1, 2});
  /* Equivalently:
  final model MyTF = TransferFunction(final a, final b = {1, 2});
  */
  MyTF tf1;                          // fine
  MyTF tf2(a = {1, 2});              // error, all elements in MyTF are final
  model M = MyTF(a = {4});           // error, all elements in MyTF are final
  model TFX
    extends MyTF;                    // fine
    Real foo = 1.0;
  end TFX;
  TFX tfx(foo = 2.0);                // fine, foo is not from MyTF
  TFX tfx2(a = {1, 3});              // error, all elements from MyTF are final
  model TFX3 = TFX(a = {1, 4});      // error, all elements from MyTF are final
end Test2;
```

]

7.2.7 Removing Modifiers – break

Modifications may contain the special keyword **break** instead of an expression. The intention of **break** is to remove the value.

The modifiers using **break** are merged using the same rule as other modifications, and follow the same restrictions so they cannot override a final modifier. During flattening of an instantiated model, remaining **break** modifications (i.e., the ones that are not further overriden) are treated as if the expression was missing. The **break** modifier for a variable of a simple type can be applied to the value and/or to specific attributes. Unless **final** was specified, it is possible to override even if no value is present, either because there was no expression originally or because **break** overrides another **break**.

[*In a dialog, a tool may hide the keyword* **break** *and show an empty input field, without the overriden modification. It should also be possible to remove this modifier to restore the overriden modification.*

There are also other uses of the keyword **break**, *but importantly it is not an expression and thus it cannot be used as a sub-expression.*]

[*Example: Remove unwanted defaults for parameters:*

```
partial model PartialStraightPipe
  parameter Real roughness = 2.5e-5 "Average height of surface
   asperities";
  parameter Real height_ab (unit = "m" ) = 0 "Height between a and b";
  ...
end PartialStraightPipe;

model StaticPipe
```

```
  extends PartialStraightPipe;
  parameter Real p_a_start = system.p_start;
  ...
end StaticPipe;

model MyPipe "Without defaults"
  extends StaticPipe(
    p_a_start = break,
    roughness = break,
    height_ab = break);
end MyPipe;
```

Replace a given parameter value by an initial computation:

```
model A
  parameter Real diameter  = 1;
  final parameter Real radius = diameter / 2;
end A;

model B "Initial equation for diameter"
  extends A( final diameter(fixed = false) = break );
  parameter Real square=2;
initial equation
  //  solving equation below for diameter
  square = f(diameter);
end B;
```

Replace the value for an inherited variable with a value computed from an algorithm:

```
model A
  Real x = 1;
end A;

model B "Computing x instead"
  extends A(final x=break);
algorithm
  x:=0;
  while ...
    x := x + ...
  end while;
end B;
```

Note that this is only legal because the modifier is modifying an inherited declaration. Due to section 4.8 it is not legal to construct the corresponding component declaration, `A a(x=break);`.]

7.3 Redeclaration

A **redeclare** construct in a modifier replaces the declaration of a local class or component with another declaration. A **redeclare** construct as an element replaces the declaration of a local class or component with another declaration. Both **redeclare** constructs work in the same way. The **redeclare** construct as an element requires that the element is inherited, and cannot be combined with a modifier of the same element in the **extends**-clause. For modifiers, the redeclare of classes uses the *short-class-definition* construct, which is a special case of normal class definitions and semantically behaves as the corresponding *class-definition*.

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 prefixes 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 prefix within one of the following groups is present in the new declaration the old prefixes 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, stream
- discrete, parameter, constant
- input, output
- array dimensions

Note that if the old declaration was a short class definition with array dimensions the array dimensions are not automatically preserved, and thus have to be repeated in the few cases they are used.

Replaceable component array declarations with array sizes on the left of the component are seen as syntactic sugar for having all arrays sizes on the right of the component; and thus can be redeclared in a consistent way.

The presence of annotations on the **redeclare** construct in a modifier is deprecated, but since none of the annotations in the specification ever had a meaning in this context it only impacts vendor-specific annotations.

[*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.*]

[

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

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

Example of arrays on the left of the component name:

```
model M
  replaceable Real [4] x[2];
  // Seen as syntactic sugar for "replaceable Real x[2, 4];"
  // Note the order.
end M;
M m(redeclare Modelica.Units.SI.Length x[2, 4]); // Valid redeclare of the type
```

]

7.3.1 The "class extends" Redeclaration Mechanism

A class declaration of the type **redeclare 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. Inherited

`B` here means that the class containing **`redeclare class extends`** `B(...)` should also inherit another declaration of `B` from one of its **`extends`**-clauses. The new declaration should explicitly include **`redeclare`**.

[*Since the rule about applying the optional class-modification implies that all declarations are inherited with modifications applied, there is no need to apply modifiers to the new declaration.*]

For **`redeclare class extends`** `B(...)` the inherited class is subject to the same restrictions as a redeclare of the inherited element, and the original class `B` should be *replaceable*, and the new element is only replaceable if the new definition is replaceable. In contrast to normal extends it is not subject to the restriction that `B` should be transitively non-replaceable (since `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 4.6):

```
class-definition :
   [ encapsulated ] class-prefixes
   class-specifier

class-specifier : long-class-specifier | ...

long-class-specifier : ...
    | extends IDENT [ class-modification ] description-string
      composition end IDENT
```

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 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
  redeclare 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 dynamicViscosity;
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. At a first glance an alternative exists that is more straightforward and 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;

  function 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) ensures 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. The precedence order is that declaration modifiers override constraining type modifiers.

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.
- The modifiers for subsequent redeclarations and constraining type are the modifiers on the component or *`short-class-definition`* if that is used in the original declaration, otherwise empty.

The syntax of a *`constraining-clause`* is as follows:

```
constraining-clause :
   constrainedby name [ class-modification ]
```

[*Example: Merging of modifiers:*

```
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 is equivalent to defining `D` *as*

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

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

```
model ElectricalSource
  replaceable SineSource source constrainedby M0(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;
```

`Circuit2` *is intended to illustrate that a user can still select any resistor model (including the original one, as is done in* `Circuit3`*), since the constraining type is kept from the original declaration if not specified in the redeclare. Thus it is easy to select an advanced resistor model, without limiting the possible future changes.*

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. Similarly the type used in a redeclaration must have the same number of dimensions as the type of redeclared element.

[*Example:*

```
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). Thus if* `T2` *is a scalar type (e.g.,* `type T2 = Real`*) then* `T1` *must also be a scalar type, and if* `T2` *is defined as vector type (e.g.,* `type T2 = Real[3]`*) then* `T1` *must also be vector type.*]

7.3.2.1 Constraining-Clause Annotations

Description and annotations on the *`constraining-clause`* are applied to the entire declaration, and it is an error if they also appear on the definition.

[*The intent is that the description and/or annotation are at the end of the declaration, but it is not straightforward to specify this in the grammar.*]

[*Example:*

```
replaceable model Load1 =
  Resistor constrainedby TwoPin "The Load"; // Recommended
replaceable model Load2 =
  Resistor "The Load" constrainedby TwoPin; // Identical to Load1
replaceable model Load3 =
  Resistor "The Load" constrainedby TwoPin "The Load"; // Error

replaceable Resistor load1
  constrainedby TwoPin "The Load"; // Recommended
replaceable Resistor load2
  "The Load" constrainedby TwoPin; // Identical to load1
replaceable Resistor load3
  "The Load" constrainedby TwoPin "The Load!"; // Error
```

]

See also the examples in section 7.3.4.

7.3.3 Restrictions on Redeclarations

The following additional constraints apply to redeclarations (after prefixes 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`**.

  [*Redeclaration with the same type can be used to restrict variability and/or change array dimensions.*]

- An element declared as `constant` cannot be redeclared.
- An element declared as `final` shall 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; provided the sub-typing rules in section 6.4 are satisfied.

  [*This is one example of redeclaration of non-replaceable elements.*]

7.3.4 Annotations for Redeclaration and Modification

A declaration can have an annotation `choices` containing modifiers on `choice`, where each of them indicates a suitable redeclaration or modifications of the element. Lookup inside a `choice` modifier is performed in the context of the annotation, meaning that references may need to be transformed to preserve the meaning when a `choice` is applied in a different context.

[*It is recommended to avoid expressions with references to elements that are not globally accessible, such as contents within a* **`protected`** *section of a class. By starting names with a dot it can be ensured that no transformation of references will be needed when a* `choice` *is applied, and that applicability of a* `choice` *does not depend on context, see section 5.3.3.*]

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.

It is allowed to include choices that are invalid in some contexts, e.g., a value might violate a **min**-attribute. (Options for tools encountering such choices include not showing them, marking them as invalid, or detecting the violations later.)

For a `Boolean` variable, a **choices** annotation may contain the definition **checkBox** = `true`, meaning to display a checkbox to input the values `false` or `true` in the graphical user interface.

The annotation **choicesAllMatching** = `true` on the following kinds of elements indicates that tools should automatically construct a menu with appropriate choices.

- For a replaceable element the included elements should be usable for replacing it. Exact criteria for inclusion in such a menu are not defined, but there shall be a a way to at least get a selection of classes, **A.B.....X.Z**, that are either directly or indirectly derived by inheritance from the constraining class of the declaration, where **A** to **X** are non-partial packages, and **Z** is non-partial.
- For a record variable the included elements shall include matching record constants and calls of matching record constructors (matching classes as for replaceable elements).

This menu can be disabled using annotation **choicesAllMatching** = `false`. It is possible to combine the two annotations for one declaration, and tools may avoid generating duplicate menu entries in that case.

[*When* **choicesAllMatching** *is not specified the following behavior is recommended for replaceable elements. A tool could ideally present (at least) the same choices as for* **choicesAllMatching** = `true`*, but if it takes (too long) time to present the list it might be better to use the* **choicesAllMatching** = `false` *behavior instead.*]

[*Example: Demonstrating the* **choices** *and* **choicesAllMatching** = `true` *annotations applied to replaceable elements.*

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

replaceable package Medium = Modelica.Media.Water.ConstantPropertyLiquidWater
  constrainedby Modelica.Media.Interfaces.PartialMedium
  annotation(choicesAllMatching = true);
```

]

[*Example: Demonstrating the* **choicesAllMatching** = `true` *annotation for parameter records.*

```
record Medium
  parameter SI.Density rho "Density";
  ...
end Medium;

record Air_30degC = Medium(rho = 1.149, ...);
constant Medium MyAir = Medium(rho = 1.1, ...);
```

```
model OpenTank
  parameter Medium medium = Medium() annotation(choicesAllMatching = true);
end OpenTank;
```

The choices for `medium` *shall include* `Medium()`, `Air_30degC()`, *and* `MyAir`. *If* `Medium()` *is chosen it is necessary to also set its* `rho`*-parameter.*]

[*Example: Applying the* `choices` *annotation 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
  parameter KindOfController x;
end A;
A a(x = 3 "PID");
```

The `choices` *annotation can also be applied to* `Boolean` *variables to define a check box.*

```
parameter Boolean useHeatPort = false annotation(choices(checkBox = true));
```

]

7.4 Selective Model Extension

[*The goal of selective model extension is to enable unforeseen structural variability without requiring deliberately prepared base-models, Bürger (2019).*

This is done by deselecting specific elements from a base class, described here, combined with adding elements as normal.]

Selective model extension is activated by using one (or more) *`inheritance-modification`* in the optional *`class-or-inheritance-modification`* of an **`extends`**-clause.

[*There is no corresponding mechanism for component modifications, short class definitions, or constrainedby.*]

Consider a class `C` with an **`extends`**-clause deselecting D:

```
model C
  extends B(..., break D, ...);
  ...
end C;
```

The semantic rules are:

1. The deselection `break` D is applied before any other, non selective model extension related, modifications of `B` in `C`.
2. When adding elements from `B` to `C` the elements matched by any deselection in **`extends`** `B` are excluded.
   - A component deselection, `break f`, matches the component with that name, `f`, of `B` and all connections with the component or its subcomponents. Matched components must be models, blocks or connectors.
   - A connection deselection, `break connect(a, b)`, matches all syntactical equivalent connections of `B`. A connection `connect(c, d)`, with `c` and `d` arbitrary but valid connection arguments, is syntactically equivalent to a connection deselection `break connect(a, b)`, if, and only if, either, `c` is syntactically equivalent to `a` and `d` is syntactically equivalent to `b` or, vice versa, `c` is syntactically equivalent to `b` and `d` is syntactically equivalent to `a`. Two code

fragments `a` and `c` are syntactically equivalent, if, and only if, the context-free derivations of `a` and `c` according to the grammar given in appendix A.2.7 are the same.

3. Conditionally declared components of `B` are assumed to be declared for all purposes of matching.
4. The deselection **break** *D* must match at least one element of `B`.
5. The component deselection are applied before the connection deselections of the same **extends**-clause.

[*Example: The following gives three typical use cases: adding a component* on a connection, *replacing a non-replaceable component, and finally constructing a reusable model from an example.*

```
model System "An example model"
  Plant plant;
  BearingFriction friction;
  Controller controller;
  StepReference reference;
equation
  connect(reference.y, controller.u_s);
  connect(plant.y, controller.u_m);
  connect(controller.y, plant.u);
  connect(friction.flange_a, plant.flange_a);
end System;

model FilterMeasurement "Component on a connection"
  extends System(break connect(plant.y, controller.u_m));
  BesselFilter filter;
equation
  connect(plant.y, filter.u);
  connect(filter.y, controller.u_m);
end FilterMeasurement;

model SampledControllerSystem "Replacing non-replaceable"
  extends System(break controller);
  SampledController controller;
equation
  connect(reference.y, controller.u1); // Note: Different name
  connect(plant.y, controller.u_m);
  connect(controller.y, plant.u);
end FilterMeasurement;

model NewPlant "Reusable model from example"
  extends System(break controller, break reference);
  RealInput u;
  RealOutput y;
equation
  connect(u, plant.u);
  connect(plant.y, y);
end NewPlant;
```

In these examples it would be possible to modify the `System` *model instead, but in many cases that is not realistic. For instance, it may not be possible to modify the* `System` *and the controlled system may be comprised of a large number of components in* `System` *– instead of only two.*]

[*Some consequences of the rules are listed below:*

The syntax ensures that nested components cannot be deselected.

Deselected components cannot be modified, neither in the **extends***-clause nor when using* `C`*. However,* `C` *may add a component with same name as a deselected component (directly or through another* **extends** `-clause`*) and that new component can be modified when using* `C`*.*

A class using selective model extension is not necessarily a sub-type of its base class.

Deselection is designed to be light-weight in particular:

- *Deselection is independent of any modification.*
- *What is deselected can be determined without considering any modifications, neither of the extending class* `C` *nor its base class* `B`.
- *There is no need to instantiate any classes to know that some component is deselected (i.e., not there) for every possible instance of the model with the deselection. An instance tree is not required.*
- *Selective model extension operates on the syntactic level only.*
- *Conditional components can be deselected without evaluating whether they are disabled or not. In particular deselecting a disabled conditional component is not an error. Connections involving the deselected conditional component are by the deselection removed as for a disabled component.*
- *Assuming the deselections are semantically valid they can be handled in any order. Handling component deselections before connection deselections is only necessary to semantically check that a connection deselection does not involve a deselected component.*

[Example: The syntactic equivalence of connection deselection ensures that connect-statements in for-loops can be deselected:

```
model B
  ...
equation
  if b then
    for i in 2:10 loop
      connect( // This comment does not impact syntactic equivalence.
        a[i],
        b[2*i] /* Without whitespace in the indexing expression. */ );
    end for;
  else
    for i in 20:30 loop
      connect(b[i], a[2*i]);
    end for;
  end for;
end B;
model C
  extends B(break connect(b[2 * i], a[i]);
end C;
```

In this case the deselection removes all of the connect-statements.]

]

Chapter 8

Equations

An *equation* is 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 nth variable can be inferred (solved for). In contrast to an algorithm section, there is no order between the equations in an equation section and they can be solved separately.

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*, which include both declaration equations and element modification for the value of the variable itself. These are considered equations when appearing outside functions, and then a component with a binding equation has its value bound to some expression. (Binding equations can also appear in functions, see section 12.4.4.)
- *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

An equation section is comprised of the keyword **`equation`** followed by a sequence of equations. The formal syntax is as follows:

```
equation-section :
   [ initial ] equation { equation ";" }
```

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-equation
     | when-equation
     | component-reference function-call-args
   )
   description
```

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 6.7).

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 ";"
```

A **for**-equation may optionally use several iterators (*for-indices*), see section 11.2.2.3 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
```

8.3.2.1 Explicit Iteration Ranges of For-Equations

The `expression` of a **for**-equation shall be a vector expression, where more general array expressions are treated as vector of vectors or vector of matrices. 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 evaluable. The iteration range of a **for**-equation can also be specified as `Boolean` or as an enumeration type, see section 11.2.2.2 for more information. The loop-variable (`IDENT`) is in scope inside

the loop-construct and shall not be assigned to. For each element of the evaluated vector expression, in the normal order, the loop-variable gets the value of that element and that is used to evaluate the body of the **for**-loop.

[*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  // j takes the values 1, 2, 3, 4
    x[j] = j;        // Uses the loop-variable j
  end for;
```

]

8.3.2.2 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];
equation
  for i loop          // Same as: for i in 1:size(x, 1) loop
    x[i] = 2 * y[i];
  end for;
```

]

8.3.3 Connect-Equations

A `connect`-equation has the following syntax:

```
connect "(" component-reference "," component-reference ")" ";"
```

These can be placed inside **for**-equations and **if**-equations; provided the indices of the **for**-loop and conditions of the **if**-equation are parameter expressions that do not depend on `cardinality`, `rooted`, `Connections.rooted`, or `Connections.isRoot`. The **for**-equations/**if**-equations are expanded. `connect`-equations are described in detail in section 9.1.

The same restrictions apply to `Connections.branch`, `Connections.root`, and `Connections.potentialRoot`; which after expansion are handled according to section 9.4.

8.3.4 If-Equations

The **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.

The **if**-equations in equation sections which do not have exclusively parameter expressions as switching conditions shall have the same number of equations in each branch (a missing else is counted as zero equations and the number of equations is defined after expanding the equations to scalar 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

The **when**-equations have the following syntax:

```
when expression then
  { equation ";" }
{ elsewhen expression then
  { equation ";" }
}
end when ";"
```

The **expression** of a **when**-equation shall be a discrete-time Boolean scalar or vector expression. If **expression** is a clocked expression, the equation is referred to as a *clocked* **when**-*clause* (section 16.6) rather than a **when**-equation, and is handled differently. The equations within a **when**-equation are activated only at the instant 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;
```

Here, `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 value of the special functions **initial**, `terminal`, and `sample` is `false`.

8.3.5.2 Where a When-Equation May Occur

- **when**-equations shall not occur inside initial equations.
- **when**-equations cannot be nested.
- **when**-equations can only occur within **if**-equations and **for**-equations if the controlling expressions are exclusively parameter expressions.

[*Example: The following* **when***-equation is invalid:*

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

]

8.3.5.3 Equations within When-Equations

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`.
- The **for**- and **if**-equations if the equations within the **for**- and **if**-equations satisfy these requirements.

Additionally,

- The different branches of **when**/**elsewhen** must have the same set of component references on the left-hand side. Here, the destination variable of a `reinit` (including when inside a **when**-clause activated with **initial**()) is not considered a left-hand side, and hence `reinit` is unaffected by this requirement (as are `assert` and `terminate`).
- The branches of an **if**-equation inside **when**-equations must have the same set of component references on the left-hand side, unless all switching conditions of the **if**-equation are parameter expressions.
- Any left hand side reference, (`v`, `out1`, ...), in a **when**-clause must be a component reference, and any indices must be parameter expressions.

[*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.*]

[*Example: The restrictions for* **if***-equations mean that both of the following variants are illegal:*

```
  Real x, y;
equation
  if time < 1 then
    when sample(1, 2) then
      x = time;
    end when;
  else
    when sample(1, 3) then
      y = time;
    end when;
  end if;

  when sample(1, 2) then
    if time < 1 then
      y = time;
    else
      x = time;
    end if;
  end when;
```

whereas the restriction to parameter-expression is intended to allow:

```
  parameter Boolean b = true;
  parameter Integer n = 3;
  Real x[n];
equation
  if b then
    for i in 1 : n loop
      when sample(i, i) then
        x[i] = time;
      end when;
    end for;
  end if;
```

]

8.3.5.4 Single Assignment Rule Applied to When-Equations

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

- Two **when**-equations shall *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.*]

- A **when**-equation 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!
equation
  ...
  when condition1 then
    close = true;
  elsewhen condition2 then
    close = false;
  end when;
  ...
end WhenPriority;
```

  An alternative to **elsewhen** *(in an equation or algorithm) is to use an algorithm with multiple* **when***-statements. However, both statements will be executed if both conditions become* `true` *at the same time. Therefore they must be in reverse order to preserve the priority, and any side-effect would require more care.*

```
model WhenPriorityAlg
  Boolean close;   // Correct model: close defined by two when-statements!
algorithm
  ...
  when condition2 then
    close := false;
  end when;
  when condition1 then
    close := true;
  end when;
  ...
end WhenPriorityAlg;
```

]

8.3.6 reinit

`reinit` can only be used in the body of a **when**-equation. It has the following syntax:

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

The operator reinitializes **x** with **expr** at an event instant. **x** is a component-reference (where any subscripts are evaluable) referring to a **Real** variable (or an array of **Real** variables) that must be selected as a state (resp., states), i.e., **reinit** on **x** implies **stateSelect = StateSelect.always** on **x**. **expr** needs to be type-compatible with **x**. For any given variable (possibly an array variable), **reinit** can only be applied (either to an individual variable or to a part of an array variable) in one **when**-equation (applying **reinit** to a variable in several **when**- or **elsewhen**-clauses of the same **when**-equation is allowed). If there are multiple **reinit** for a variable inside the same **when**- or **elsewhen**-clause, they must appear in different branches of an **if**-equation (in order that at most one **reinit** for the variable is active at any event). In case of **reinit** active during initialization (due to **when initial()**), see section 8.6.

reinit does not break the single assignment rule, because **reinit(x, expr)** in equations evaluates **expr** to a value, then at the end of the current event iteration step it assigns this value to **x** (this copying from values to reinitialized state(s) is done after all other evaluations of the model and before copying **x** to **pre(x)**).

[*Example: 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** *are selected as states at least when the corresponding* **when***-clauses become active. If this is not possible, an error occurs, since otherwise* **reinit** *would be applied to a non-state variable.*

Example for the usage of **reinit** *(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 one of the following forms is an assertion:

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

Here, **condition** is a **Boolean** expression, **message** is a **String** expression, and **assertionLevel** is an optional evaluable expression of the built-in enumeration type **AssertionLevel**. It can be used in equation sections or algorithm sections.

[*This means that* **assert** *can be called as if it were a function with three formal parameters, the third formal parameter has the name* **level** *and the default value* **AssertionLevel.error**.]

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. If the simulation is aborted, **message** indicates the cause of the error.

  [*Ways to continue simulation with another evaluation include using a shorter step-size, or changing the values of iterationvariables.*]

  Failed assertions take 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.

`terminate` takes a string argument indicating the reason for the success.

[*Example: The intention of* `terminate` *is to give more complex stopping criteria than a fixed point in time:*

```
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. Discrete-time variables keep their values until these variables are explicitly changed. Differentiated variables have `der(x)` corresponding to the time-derivative of `x`, and `x` is continuous, except when `reinit` is triggered, see section 8.3.6. 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 equations express relations between variables which have to be fulfilled concurrently.
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. There must exist a perfect matching of variables to equations after flattening, where a variable can only be matched to equations that can contribute to solving for the variable (*perfect matching rule* – previously called *single assignment rule*); see also globally balanced section 4.8.

8.5 Events and Synchronization

An *event* is 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`**-equation, or **`if`**-expression.

The integration is halted and an event occurs whenever an event generation expression, e.g., `x > 2` o or `floor(x)`, changes its value. An event generating expression has an internal buffer, and the value of the expression can only be changed at event instants. If the evaluated expression is inconsistent with the buffer, that will trigger an event and the buffer will be updated with a new value at the event instant. During continuous integration event generation expression has the constant value of the expression from the last event instant.

[*A root finding mechanism is needed which determines a small time interval in which the expression changes its value; the event occurs at the right side of this interval.*]

[*Example:*

```
y = if u > uMax then uMax else if 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 `noEvent`. `smooth` also allows relations used as argument to be taken literally. The `noEvent` feature is propagated to all subrelations in the scope of the `noEvent` application. For `smooth` the liberty to not allow literal evaluation is propagated to all subrelations, but the smoothness 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` *operator is useful, if, e.g., the modeler can guarantee that the used* **`if`***-expressions 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` *operator 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 `noEvent`, 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: Two different errors caused by non-discrete-time expressions:*

```
when noEvent(x1 > 1) or x2 > 10 then // When-condition must be discrete-time
  close = true;
end when;
above1 = noEvent(x1 > 1);            // Boolean equation must be discrete-time
```

The when-condition rule is stated in section 8.3.5, and the rule for a non-`Real` *equation is stated in section 3.8.5.*]

Modelica is based on the synchronous data flow principle (section 8.4).

[*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;
  elsewhen 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, called the *initialization problem*, 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 `initial()`, and only in one of the two forms **when** `initial()` **then** or **when** `{..., initial(), ...}` **then** (and similarly for **elsewhen** and algorithms see below). In this case, the **when**-clause equations remain active during the whole initialization phase. In case of a `reinit(x, expr)` being active during initialization (due to being inside **when** `initial()`) this is interpreted as adding `x = expr` (the `reinit` -equation) as an initial equation. The `reinit` handling applies both if directly inside **when**-clause or inside an **if**-equation in the **when**-clause. In particular, `reinit(x, expr)` needs to be counted as the equation `x = expr;` for the purpose of balancing of **if**-equations inside **when**-clauses that are active during initialization, see section 8.3.4.

[*If a* **when***-clause equation* `v = expr;` *is not active during the initialization phase, the equation* `v = pre(v)` *is added for initialization. This follows from the mapping rule of* **when***-clause equations. If the condition of the* **when***-clause contains* `initial()`*, but not in one of the specific forms, the* **when***-clause is not active during initialization:* **when** `not initial()` **then** `print("simulation started");` **end when**;]

The algorithmic statements within a **when**-statement are active during initialization, if and only if they are explicitly enabled with `initial()`, and only in one of the two forms **when** `initial()` **then** or **when** `{..., initial(), ...}` **then**. In this case, the algorithmic statements within the **when**-statement remain active during the whole initialization phase.

An active **when**-clause inactivates the following **elsewhen** (similarly as for **when**-clauses during simulation), but apart from that the first **elsewhen** `initial()` **then** or **elsewhen** `{..., initial(), ...}` **then** is similarly active during initialization as **when** `initial()` **then** or **when** `{..., initial(), ...}` **then**.

[*That means that any subsequent* **elsewhen** `initial()` *has no effect, similarly as* **when** `false` **then**.]

[*There is no special handling of inactive* **when***-statements during initialization, instead variables assigned in* **when***-statements are initialized using* `v := pre(v)` *before the body of the algorithm (since they are discrete), see section 11.1.2.*]

Further constraints, necessary to determine the initial values of all variables (depending on the component variability, see section 4.5 for definitions), can be defined in the following 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. For a continuous-time `Real` variable `vc`, the equation `pre(vc) = vc` is added to the initialization equations.

   [*If* `pre(vc)` *is not present in the flattened model, a tool may choose not to introduce this equation, or if it was introduced it can eliminate it (to avoid the introduction of many dummy variables* `pre(vc)`*).*]

3. Implicitly by using the `start`-attribute for variables with `fixed = true`. With `start` given by `startExpression`:

   - For a variable declared as `constant` or `parameter`, no equation is added to the initialization equations.
   - For a discrete-time variable `vd`, the equation `pre(vd) = startExpression` is added to the initialization equations.
   - For a continuous-time `Real` variable `vc`, the equation `vc = startExpression` is added to the initialization equations.

Constants shall be determined by declaration equations (see section 4.5.1), and `fixed = false` is not allowed. For parameters, `fixed` defaults to `true`. For other variables, `fixed` defaults to `false`.

`start`-values of variables having `fixed = false` can be used as initial guesses, in case iterative solvers are used in the initialization phase.

[*In case of iterative solver failure, it is recommended to specially report those variables for which the solver needs an initial guess, but where the fallback value (see section 4.9) has been applied, since the lack of appropriate initial guesses is a likely cause of the solver failure.*]

If a parameter has a value for the `start`-attribute, does not have `fixed = false`, and neither has a binding equation nor is part of a record having a binding equation, the value for the `start`-attribute can be used to add a parameter binding equation assigning the parameter to that `start` value. In this case a diagnostic message is recommended in a simulation model.

[*This is used in libraries to give rudimentary defaults so that users can quickly combine models and simulate without setting parameters; but still easily find the parameters that should be set properly.*]

All variables declared as `parameter` having `fixed = false` are treated as unknowns during the initialization phase, i.e., there must be additional equations for them – and the `start`-value can be used as a guess-value during initialization.

[*In the case a parameter has both a binding equation and* `fixed = false` *a diagnostic is recommended, but the parameter should be solved from the binding equation.*

Continuous-time `Real` *variables* `vc` *have exactly one initialization value since the rules above assure that during initialization* `vc = pre(vc) = vc.startExpression` *(if* `fixed = true`*).*

Before the start of the integration, it must be guaranteed that for all variables `v`, `v = pre(v)`*. If this is not the case for some variables* `vi`, `pre(vi) := vi` *must be set and an event iteration at the initial time must follow, so the model is re-evaluated, until this condition is fulfilled.*

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.]

These non-normative considerations are addressed as follows. A tool may add or remove initial equations automatically according to the rules below such that the resulting system is structurally nonsingular:

- A missing initial value of a discrete-time variable (see section 4.5 – this does not include parameter and constant variables) 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 `pre` is not explicitly used on these variables, do not have an effect on the simulation.
- A `start`-attribute that is not fixed may be treated as fixed with a diagnostic.
- A consistent start value or initial equation may be removed with a diagnostic.

[*The goal is to be able to initialize the model, while satisfying the initial equations and fixed start values.*]

[*Example: 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;
```

]

[*Example: 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;
```

]

[*Example: 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;
```

]

[*Example: Resettable 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 and reset value for y, if not steadyState";
  input Boolean reset "For resetting integrator to y0";
  Real y;
equation
  der(y) = a * y + b * u;
  when {initial(), reset} then
    if not (initial() and steadyState) then
      reinit(y, y0);
    end if;
  end when;
initial equation
  if steadyState then
    der(y) = 0;
  end if;
```

If `not steadyState` *this will add* `y = y0` *during the initialization; if not the* `reinit` *is ignored during initialization and the initial equation is used. This model can be written in various ways, this particular way makes it clear that the reset is equal to the normal initialization.*

During initialization this gives the following equations

```
  if not steadyState then
    y = y0;
  end if;
  if steadyState then
    der(y) = 0;
  end if;
```

if `steadyState` *had not been a parameter-expression both of those equations would have been illegal according to the restrictions in section 8.3.4.*]

8.6.1 Equations Needed for Initialization

[*In general, for the case of a pure (first order) ordinary differential equation (ODE) system with n state variables and m output variables, we will have $n + m$ unknowns during transient analysis. The ODE initialization problem has n additional unknowns corresponding to the derivative variables. During 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 transient analysis.*]

[*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, `x1` *and* `x2`*, 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 initialization problem requires finding initial values for* `x1`*,* `x2`*,* `der(x1)`*,* `der(x2)`*, and* `y`*. Two additional initial equations thus need to be provided to obtain a globally balanced initialization problem. Additionally, those two initial equations must be chosen with care to ensure that they, in combination with the dynamic equations, give a well-determined 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.]

8.6.2 Start Value Recommended Priority

In general many variables have `start`-attributes that are not fixed and selecting a subset of these can give a consistent set of start values close to the user-expectations. The following gives a non-normative procedure for finding such a subset.

[*A model has a hierarchical component structure. Each component of a model can be given a unique model component hierarchy level number. The top-level model has a level number of 1. The level number increases by 1 for each level down in the model component hierarchy. The model component hierarchy level number is used to give* `start`*-attribute a confidence number, where a lower number means that the* `start`*-attribute is more confident. Loosely, if the* `start`*-attribute is set or modified on level i then the confidence number is i. If a* `start`*-attribute is set by a possibly hierarchical modifier at the top level, then this* `start`*-attribute has the highest confidence, namely 1 irrespectively on what level, the variable itself is declared. If the* `start`*-attribute is set equal to a parameter, which may be equal to another parameter (etc), the lowest confidence number of these bindings are used. (In almost all cases that is the confidence number of the last parameter binding in the chain.) Note that this is only applied if the expression is exactly the parameter – not an expression depending on one or more parameters. In case the confidence number considering parameter bindings is tied the confidence number of the* `start`*-attribute is used to break the tie, if unequal.*

[Example: Simplified examples showing the priority of start-values. The example `M3` *shows that it is important that parameter-confidence is used directly and not only when the other priority is tied.*

```
model M1
  Real x(start = 4.0);
  Real y(start = 5.0);
equation
  x = y;
end M1;
model M2
  parameter Real xStart = 4.0;
  parameter Real yStart = 5.0;
  Real x(start = xStart);
  Real y(start = yStart);
equation
  x = y;
end M2;
model M3
  model MLocal
    parameter Real xStart = 4.0;
    Real x(start = xStart);
  end MLocal;
  model MLocalWrapped
    parameter Real xStart = 4.0;
    MLocal m(xStart = xStart);
  end MLocalWrapped;
  MLocal mx;
  MLocalWrapped my(xStart = 3.0);
equation
  mx.x = my.y;
end M3;
M1 m1(x(start = 3.0));
// Using m1.x.start = 3.0 with confidence number 1
// over m1.y.start = 5.0 with confidence number 2
M2 m2(xStart = 3.0);
// Using m2.x.start = m2.xStart = 3.0 with confidence number 1
// over m2.y.start = m2.yStart = 5.0 with confidence number 2
M3 m3;
// Using m3.my.x = m3.my.xStart = 3.0 with confidence number 1
// over m3.mx.x = m3.mx.xStart = 4.0 with confidence number 2
```

]
]

Chapter 9

Connectors and Connections

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

Connectors and **connect**-equations are designed so that different components can be connected graphically with well-defined semantics. However, the graphical part is optional and found in chapter 18.

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 ")" ";"
```

[*A* connector *is an instance of a* **connector** *class.*]

The **connect**-equation construct takes two references to connectors, each of which is either of the following forms:

- $c_1.c_2.\ldots.c_n$, where c_1 is a connector of the class, $n \geq 1$ and c_{i+1} is a connector element of c_i for $i = 1, \ldots, (n-1)$.
- `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 or the special operator `:`. 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: Array usage:*

```
  connector InPort = input Real;
  connector OutPort = output Real;
  block MatrixGain
    input InPort u[size(A, 2)];
    output OutPort y[size(A, 1)];
    parameter Real A[:, :] = [1];
  equation
    y = A * u;
  end MatrixGain;
  Modelica.Blocks.Sources.Sine sinSource[5];
  MatrixGain gain (A = 5 * identity(5));
  MatrixGain gain2(A = ones(2, 5));
  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 (*connection equations*).

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`. Any other connector elements that is hierarchically inside `M`, but not in one of the outside connectors of `M`, is called an *inside connector* with respect to `M`. This is done before resolving **`outer`** elements to corresponding **`inner`** ones.

[*Example:*

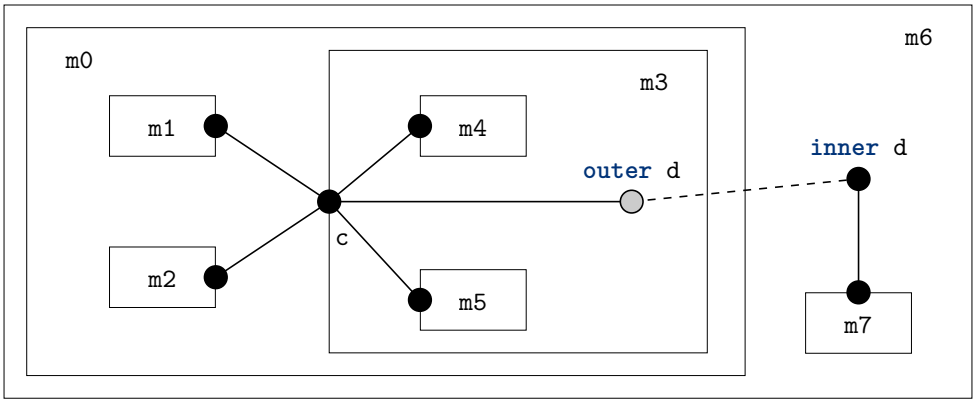


Figure 9.1: *Example for inside and outside connectors.*

The figure visualizes the following `connect`*-equations to the connector* `c` *in the models* `mi`*. Consider the following* `connect`*-equations found in the model for component* `m0`*:*

```
connect(m1.c, m3.c); // m1.c and m3.c are inside connectors
connect(m2.c, m3.c); // m2.c and m3.c are inside connectors
```

and in the model for component `m3` *(*`c.x` *is a sub-connector inside* `c`*):*

```
connect(c, m4.c);   // c is an outside connector, m4.c is an inside connector
connect(c.x, m5.c); // c.x is an outside connector, m5.c is an inside connector
connect(c, d);      // c is an outside connector, d is an outside connector
```

and in the model for component `m6`*:*

```
connect(d, m7.c);   // d is an outside connector, m7.c is an inside 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*, non-parameter scalar variables and non-parameter array elements declared in expandable connectors are marked as only being potentially present. A non-parameter array element may be declared with array dimensions `:` indicating that the size is unknown (note that the size of such a dimension cannot be determined using `size`, see section 10.3.1). This applies to both variables of simple types, and variables of structured types.

Then connections containing expandable connectors are elaborated:

1. If a `connect`-equation references a potentially present component *as part of the argument* it will be marked as being present, which will allow a connection to an undeclared connector inside it. The rule does not apply for the complete argument.
2. After that at least one connector in the `connect`-equation must reference a declared component.
3. If the other connector is undeclared it must be in a declared component and is handled as follows:
   - The expandable connector instance is automatically augmented with a new component having the used name and corresponding type.
   - If the undeclared component is subscripted, an array variable is created, and a connection to the specific array element is performed. Introducing elements in an array gives an array with at least the specified elements, other elements are either not created or have a default value (i.e., as if they were only potentially present, and the same note regarding the use of `size` also applies here).
   - 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 for a non-expandable connector.

     [*The general rule ensures consistency for inside and outside connectors, and handles multiple connections to the new component. In the simple case of no other connections involving these variables and the existing side referring 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.*]

     For an array the `input`/`output` property can be deduced separately for each array element.

Additionally:

- 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 variables introduced in the elaboration follow additional rules for generating connection sets (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. An augmentation set is defined as the set of connected expandable connector instances that through the elaboration will have matching variables.

  [*Example:*

```
expandable connector EngineBus
end EngineBus;

partial block Sensor
  RealOutput speed; // Output, i.e., non-input
end Sensor;
partial block Actuator
  RealInput speed; // Input
end Actuator;

model SensorWBus
  EngineBus bus;
```

```
  replaceable Sensor sensor;
equation
  connect(bus.speed, sensor.speed);
  // Provides 'speed'
end SensorWBus;
model ActuatorWBus
  EngineBus bus;
  replaceable Actuator actuator;
equation
  connect(bus.speed, actuator.speed);
  // Uses 'speed'
end ActuatorWBus;

model Engine
  ActuatorWBus actuator;
  SensorWBus sensor;
  EngineBus bus;
equation
  connect(bus, actuator.bus);
  connect(bus, sensor.bus);
end Engine;
```

This small example shows how expandable connectors are normally used:

- *There are a number of bus-instances all connected together. Often they have the same name, but it is not required.*
- *There is one source of the signal:* `sensor.sensor.speed`.
- *There are zero or more uses of the signal:* `actuator.actuator.speed`.

]

- 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 Modelica.Units.SI;
  SI.AngularVelocity speed;
  SI.Temperature T;
end EngineBus;

partial block Sensor
  RealOutput speed;
end Sensor;

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

  ]

- An expandable connector shall 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.Units.SI.Current i; // Error
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;
```

]

- Expandable connectors can only be connected to other expandable connectors.

If a `connect`-equation references a potentially present variable, or variable element, in an expandable connector the variable or variable element is marked as being present, and due to the paragraphs above it is possible to deduce whether the bus variable shall be treated as input, or shall be treated as output in the `connect`-equation. That `input` or `output` prefix is added if no `input`/`output` prefix is present on the declaration.

[*Example:*

```
expandable connector EmptyBus
end EmptyBus;

model Controller
  EmptyBus bus1;
  EmptyBus bus2;
  RealInput speed;
equation
  connect(speed, bus1.speed); // OK; only one undeclared and not subscripted.
  connect(bus1.pressure, bus2.pressure); // Error; both undeclared.
  connect(speed, bus2.speed[2]); // Introduces speed array (with element [2]).
end Controller;
```

]

An expandable connector array component for which `size` is not defined (see section 10.3.1) is referred to as a *sizeless array component*. Such a component shall not be used without subscripts, and the subscripts must slice the array so that the sizeless dimensions are removed.

[*Example: Valid and invalid uses of sizeless array components:*

```
expandable connector EngineBus
end EngineBus;

partial block Sensor
  RealOutput speed;
end Sensor;

model Engine
  parameter Integer n = 1;
  EngineBus bus;
  replaceable Sensor sensor;
  RealOutput sensorSpeeds[:];
equation
  /* Comments below refer to the use of sizeless array bus.speed. */
  connect(bus.speed[n], sensor.speed) ; // OK; subscript to scalar component.
  connect(bus.speed, sensorSpeeds); // Error; missing subscripts.
public
  Real a[:] = bus.speed; // Error; missing subscripts.
  Real b[2] = bus.speed[{1, 3}]; // OK; subscript selects fixed size sub-array.
end Engine;
```

]

After this elaboration the expandable connectors are treated as normal connector instances, and the connections as normal connections, and all potentially present variables and array elements that are not actually present are undefined. It is an error if there are expressions referring to potentially present variables or array elements that are not actually present or non-declared variables. This elaboration implies that expandable connectors can be connected even if they do not contain the same components.

[*A tool may remove undefined variables in an expandable connector, or set them to the default value, e.g., zero for* `Real` *variables.*]

[*Expressions can only "read" variables from the bus that are actually declared and present in the connector, in order that the types of the variables can be determined in the local scope.*]

[*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 Modelica.Units.SI;
import Modelica.Blocks.Interfaces.RealInput;
// Plant Side
model SparkPlug
  RealInput spark_advance;
  ...
end SparkPlug;

expandable connector EngineBus
  // No minimal set
end EngineBus;

expandable connector CylinderBus
  Real spark_advance;
end CylinderBus;

model Cylinder
  CylinderBus cylinder_bus;
  SparkPlug spark_plug;
  ...
equation
  connect(spark_plug.spark_advance,
  cylinder_bus.spark_advance);
end Cylinder;

model I4
  EngineBus engine_bus;
  Modelica.Mechanics.Rotational.Sensors.SpeedSensor speed_sensor;
  Modelica.Thermal.HeatTransfer.Sensors.TemperatureSensor temp_sensor;
  parameter Integer nCylinder = 4 "Number of cylinders";
  Cylinder cylinder[nCylinder];
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)
  for i in 1:nCylinder loop
    connect(cylinder[i].cylinder_bus,
    engine_bus.cylinder_bus[i]);
  end for;
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_bus[1];
CylinderBus cylinder_bus[2];
CylinderBus cylinder_bus[3];
CylinderBus cylinder_bus[4];
```

]

9.2 Generation of Connection Equations

When generating *connection equations*, **outer** elements are resolved to the corresponding **inner** elements in the instance hierarchy (see instance hierarchy name lookup section 5.4). The arguments to each connect-equation are resolved to two connector elements.

For every use of the connect-equation

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

a connection set is generated for each pair of corresponding primitive components of a and b together with an indication of whether they are from an inside or an outside connector.

Definition 9.1. ***Primitive elements.*** The primitive elements are of simple types or of types defined as **operator record** (i.e., a component of an **operator record** type is not split into sub-components). □

The elements of the connection sets are tuples of primitive variables together with an indication of inside or outside; if the same tuple belongs to two connection sets those two sets are merged, until every tuple is only present in one set. Composite connector types are broken down into primitive components. The **outer** components are handled by mapping the objects to the corresponding **inner** components, and the inside indication is not influenced. The outer connectors are handled by mapping the objects to the corresponding inner connectors, and they are always treated as outside connectors.

[*Rationale: The inside/outside as part of the connection sets ensure that connections from different hierarchical levels are treated separately. Connection sets are formed from the primitive elements and not from the connectors; this handles connections to parts of hierarchical connectors and also makes it easier to generate equations directly from the connection sets. All variables in one connection set will either be flow variables or non-flow variables due to restriction on* connect*-equations. The mapping from an* **outer** *to an* **inner** *element must occur before merging the sets in order to get one zero-sum equation, and ensures that the equations for the* **outer** *elements are all given for one side of the connector, and the* **inner** *element can define the other side.*]

The following connection sets with just one member are also present (and merged):

- Each primitive flow variable as inside connector.
- Each flow variable *added* during augmentation of expandable connectors, both as inside and as outside.

  [*Note that the flow variable is not directly in the expandable connector, but in a connector inside the expandable connector.*]

[*Rationale: If these variables are not connected they will generate a set comprised only of this element, and thus they will be implicitly set to zero (see below). If connected, this set will be merged and adding this at the start has no impact.*]

Each connection set is used to generate equations for potential and flow (zero-sum) variables of the form

- $a_1 = a_2 = \ldots = a_n$ (neither flow nor stream variables)
- $z_1 + z_2 + (-z_3) + \ldots + z_n = \mathbf{0}$ (flow variables)

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

For an **operator record** type this uses the operator `'0'` – which must be defined in the operator record – and all of the flow variables for the **operator record** must be of the same **operator record** type. This implies that in order to have flow variables of an **operator record** type the **operator record** must define addition, negation, and `'0'`; and these operations should define an additive group.

In order to generate equations for flow variables (using the `flow` prefix), the sign used for the connector variable z_i above is +1 for inside connectors and -1 for outside connectors (z_3 in the example above).

[*Example: Simple example:*

```
model Circuit
  Ground ground;
  Load load;
  Resistor resistor;
equation
  connect(load.p , ground.p);
  connect(resistor.p, ground.p);
end Circuit;

model Load
  extends TwoPin;
  Resistor resistor;
equation
  connect(p, resistor.p);
  connect(resistor.n, n);
end Load;
```

The connection sets are before merging (note that one part of the load and resistor is not connected):

{<load.p.i, *inside*>}
{<load.n.i, *inside*>}
{<ground.p.i, *inside*>}
{<load.resistor.p.i, *inside*>}
{<load.resistor.n.i, *inside*>}
{<resistor.p.i, *inside*>}
{<resistor.n.i, *inside*>}
{<resistor.p.i, *inside*>, <ground.p.i, *inside*>}
{<resistor.p.v, *inside*>, <ground.p.v, *inside*>}
{<load.p.i, *inside*>, <ground.p.i, *inside*>}
{<load.p.v, *inside*>, <ground.p.v, *inside*>}
{<load.p.i, *outside*>, <load.resistor.p.i, *inside*>}
{<load.p.v, *outside*>, <load.resistor.p.v, *inside*>}
{<load.n.i, *outside*>, <load.resistor.n.i, *inside*>}
{<load.n.v, *outside*>, <load.resistor.n.v, *inside*>}

After merging this gives:

{<load.p.i, *outside*>, <load.resistor.p.i, *inside*>}
{<load.p.v, *outside*>, <load.resistor.p.v, *inside*>}
{<load.n.i, *outside*>, <load.resistor.n.i, *inside*>}
{<load.n.v, *outside*>, <load.resistor.n.v, *inside*>}
{<load.p.i, *inside*>, <ground.p.i, *inside*>, <resistor.p.i, *inside*>}
{<load.p.v, *inside*>, <ground.p.v, *inside*>, <resistor.p.v, *inside*>}

*{<*load.n.i*, inside>}*
*{<*resistor.n.i*, inside>}*

And thus the equations:

```
load.p.v = load.resistor.p.v;
load.n.v = load.resistor.n.v;
load.p.v = ground.p.v;
load.p.v = resistor.p.v;
0 = (-load.p.i) + load.resistor.p.i;
0 = (-load.n.i) + load.resistor.n.i;
0 = load.p.i + ground.p.i + resistor.p.i;
0 = load.n.i;
0 = resistor.n.i;
```

]

[*Example: Outer component example:*

```
model Circuit
  Ground ground;
  Load load;
  inner Resistor resistor;
equation
  connect(load.p, ground.p);
end Circuit;

model Load
  extends TwoPin;
  outer Resistor resistor;
equation
  connect(p, resistor.p);
  connect(resistor.n, n);
end Load;
```

The connection sets are before merging (note that one part of the load and resistor is not connected):

*{<*load.p.i*, inside>}*
*{<*load.n.i*, inside>}*
*{<*ground.p.i*, inside>}*
*{<*resistor.p.i*, inside>}*
*{<*resistor.n.i*, inside>}*
*{<*load.p.i*, inside>, <*ground.p.i*, inside>}*
*{<*load.p.v*, inside>, <*ground.p.v*, inside>}*
*{<*load.p.i*, outside>, <*resistor.p.i*, inside>}*
*{<*load.p.v*, outside>, <*resistor.p.v*, inside>}*
*{<*load.n.i*, outside>, <*resistor.n.i*, inside>}*
*{<*load.n.v*, outside>, <*resistor.n.v*, inside>}*

After merging this gives:

*{<*load.p.i*, outside>, <*resistor.p.i*, inside>}*
*{<*load.p.v*, outside>, <*resistor.p.v*, inside>}*
*{<*load.n.i*, outside>, <*resistor.n.i*, inside>}*
*{<*load.n.v*, outside>, <*resistor.n.v*, inside>}*
*{<*load.p.i*, inside>, <*ground.p.i*, inside>}*
*{<*load.p.v*, inside>, <*ground.p.v*, inside>}*
*{<*load.n.i*, inside>}*

And thus the equations:

```
load.p.v = resistor.p.v;
load.n.v = resistor.n.v;
load.p.v = ground.p.v;
0 = (-load.p.i) + resistor.p.i;
```

```
0 = (-load.n.i) + resistor.n.i;
0 = load.p.i + ground.p.i;
0 = load.n.i;
```

This corresponds to a direct connection of the resistor.]

9.3 Restrictions of Connections and Connectors

- The `connect`-equations (and the special functions for overdetermined connectors) can only be used in equations, and shall not be used inside **`if`**-equations with conditions that are not parameter expressions, or in **`when`**-equations.

  [*The* **`for`***-equations always have parameter expressions for the array expression.*]

- A connector component shall not be declared with the prefix `parameter` or `constant`. In the `connect`-equation the primitive components may only connect parameter variables to parameter variables and constant variables to constant variables.

- The `connect`-equation construct only accepts forms of connector references as specified in section 9.1.

- In a `connect`-equation the two connectors must have the same named component elements with the same dimensions; recursively down to the primitive components. The primitive components with the same name are matched and belong to the same connection set.

- The matched primitive components of the two connectors must have the same primitive types, and flow variables may only connect to other flow variables, stream variables only to other stream variables, and causal variables (`input`/`output`) only to causal variables (`input`/`output`).

- A connection set of causal variables (`input`/`output`) may at most contain variables from one inside `output` connector (for state-machines extended as specified in section 17.3.6) 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 generated for a non-partial model or block:

  1. the connection set includes variables from an outside public expandable connector,
  2. the set contains variables from protected outside connectors,
  3. it contains variables from one inside `output` connector, or
  4. from one public outside `input` connector, or
  5. the set is comprised solely of one variable from one 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 (item 5 covers the case where a connector of a component is left unconnected and the source given textually).*]

- Variables from 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 shall 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.*]

- In a connection set all variables having non-empty quantity attribute must have the same quantity attribute.

- A `connect`-equation shall 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 `:`.
- Constants or parameters in connected components yield the appropriate `assert`-statements to check that they have the same value; connections are not generated.
- For conditional connectors, see section 4.4.5.

9.3.1 Balancing Restriction and Size of Connectors

For each non-partial non-simple non-expandable connector class the number of flow variables shall be equal to the number of variables that are neither `parameter`, `constant`, `input`, `output`, `stream` 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. A simple connector class is not expandable, has some time-varying variables, and has neither `input`, `output`, `stream` nor `flow` variables.

[*Expandable connector classes are excluded from this, since their component declarations are only a form of constraint.*]

A component of a simple connector class must be declared as `input`, `output`, or **`protected`**.

[*A simple connector class is thus always unbalanced, but since it is used with causality or not visible to the outside it does not unbalance the model.*]

[*Example:*

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

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

The `Frame_Illegal` *connector (intended to be used in a simple multi-body 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 connector 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.Units.SI.Position r0[3];
  input Real S[3, 3];
  input Modelica.Units.SI.Velocity v[3];
  input Modelica.Units.SI.AngularVelocity w[3];
  Modelica.Units.SI.Acceleration a[3];
  Modelica.Units.SI.AngularAcceleration z[3];
  flow Modelica.Units.SI.Force f[3];
```

```
  flow Modelica.Units.SI.Torque t[3];
end Frame_a;

connector Frame_b "correct connector"
  output Modelica.Units.SI.Position r0[3];
  output Real S[3, 3];
  output Modelica.Units.SI.Velocity v[3];
  output Modelica.Units.SI.AngularVelocity w[3];
  Modelica.Units.SI.Acceleration a[3];
  Modelica.Units.SI.AngularAcceleration z[3];
  flow Modelica.Units.SI.Force f[3];
  flow Modelica.Units.SI.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 Overconstrained Connections

There is a special problem regarding equation systems resulting from *loops* in connection graphs where the connectors contain *non-flow* (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 non-flow 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 Connection Graphs and Their Operators

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 array dimension n of `residue` shall be a constant `Integer` expression that can be evaluated during translation, with $n \geq 0$. The `equalityConstraint` expresses the equality between the two type instances `T1` and `T2` or the record instances `R1` and `R2`, respectively, as the n non-redundant equation residuals returned in `residue`. That is, the set of n non-redundant equations stating that `R1 = R2` is given by the equation (`0` represents 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. If an array is used as argument to any of the `Connections.*` functions it is treated as one unit – unlike `connect`, there is no special treatment of this case, compare section 9.1.

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 = equalityConstraint(R1, R2)` has to be used for the generation of `connect`-equations. The edges of

the virtual connection graph are implicitly defined by `connect` and explicitly by `Connections.branch`, 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.

The overconstrained equation operators for connection graphs are listed below. Here, `a` and `b` are connector instances that may be hierarchically structured, e.g., `a` may be an abbreviation for `enginePort.frame_a`.

Expression	*Description*	*Details*
`connect(a, b)`	Optional spanning-tree edge	Operator 9.1
`Connections.branch(a.R, b.R)`	Required spanning-tree edge	Operator 9.2
`Connections.root(a.R)`	Definite root node	Operator 9.3
`Connections.potentialRoot(a.R, ...)`	Potential root node	Operator 9.4
`Connections.isRoot(a.R)`	Predicate for being selected as root	Operator 9.5
`Connections.rooted(a.R)`	Predicate for being closer to root	Operator 9.6

Operator 9.1 `connect`

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

Except for redundant connections it defines an *optional spanning-tree edge* from the overdetermined type or record instances in connector `a` to the corresponding overdetermined type or record instances in connector `b` for a virtual connection graph. E.g., from `a.R` to `b.R`. The full explanation will be given in section 9.4.2. The types of the corresponding overdetermined type or record instances shall be the same.

Operator 9.2 `Connections.branch`

```
Connections.branch(a.R, b.R)
```

Defines a *required spanning-tree edge* 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 `connect`-equation is allowed.

[*E.g., it is not allowed to use this function in a* **`when`***-clause. 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, ⟨other unknowns⟩)`.]

Operator 9.3 `Connections.root`

```
Connections.root(a.R)
```

The overdetermined type or record instance `R` in connector instance `a` is a *(definite) root node* in a virtual connection graph.

[*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.*]

Operator 9.4 `Connections.potentialRoot`

```
Connections.potentialRoot(a.R)
Connections.potentialRoot(a.R, priority=p)
```

The overdetermined type or record instance `R` in connector instance `a` is a *potential root node* 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 `Connections.root` definition, one of the potential roots with the lowest priority number is selected as root.

[*This definition may be used if in a model with connector* `a` *the overdetermined record* `a.R` *appears differentiated – der(a.R) – together with the* constraint equations *of* `a.R`*, i.e., a non-redundant subset of* `a.R` *maybe used as states.*]

Operator 9.5 `Connections.isRoot`

```
Connections.isRoot(a.R)
```

Returns true, if the overdetermined type or record instance `R` in connector instance `a` is selected as a root in the virtual connection graph.

Operator 9.6 `Connections.rooted`

```
Connections.rooted(a.R)
rooted(a.R) // deprecated!
```

If the operator `Connections.rooted(a.R)` is used, or the equivalent but deprecated operator `rooted(a.R)`, then there must be exactly one `Connections.branch(a.R, b.R)` involving `a.R` (the argument of `Connections.rooted` must be the first argument of `Connections.branch`). In that case `Connections.rooted(a.R)` returns true, if `a.R` is closer to the root of the spanning tree than `b.R`; otherwise false is returned.

[*This operator can be used to avoid equation systems by providing analytic inverses, see* `Modelica.Mechanics.MultiBody.Parts.FixedRotation`.]

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

9.4.2 Generation of Connection Graph Equations

When generating connection graph equations, only the overdetermined components of a connector are considered. The connection graph equations replace the equality-equations for variables that are neither flow nor stream in section 9.2.

9.4.2.1 Handle Connect-Equation Redundancy

In order to eliminate any redundant `connect`-equation the following preparation is needed.

[*In the common case where there is no* `connect`*-equation redundancy, a consequence of this preparation is that a* `connect`*-equation between connectors with* one *overdetermined component may be directly turned into* one *optional spanning-tree edge.*]

1. The connection sets are built similarly to the normal way, but keeping the overdetermined components as primitives.
2. Instead of generating the equality-equation for an overdetermined component, an optional spanning-tree edge in the virtual connection graph is constructed.

[*If a connection set contains* n *overdetermined components, and was built from* m `connect`*-equations, then the connection set has a* `connect`*-equation redundancy of* $m-(n-1) \geq 0$. *If there is no* `connect`*-equation redundancy (i.e., if* $m = n - 1$*), the optional spanning-tree edges can be chosen to correspond to the* `connect`*-equations for overdetermined connectors. If there is a non-zero* `connect`*-equation redundancy, there will always exist* `connect`*-equations without a corresponding optional spanning-tree edge.*

It is called redundancy *since this number of* `connect`*-equations could be removed without changing the connection set or the generated equations. Situations with non-zero* `connect`*-equation redundancy include connectors connected directly to themselves, duplicated connections, and having all three pair-wise connections between the connectors* `a`, `b` *and* `c`. *The latter case can be used to consistently handle conditional components (so that disabling* `b` *does not break the connection between* `a` *and* `c`*).*]

The selected optional spanning tree edges, together with all required spanning tree edges generated from `Connections.branch`, and nodes corresponding to definite and potential roots form the virtual connection graph.

9.4.2.2 Spanning Trees

Before connection equations are generated, the virtual connection graph is transformed into a set of spanning trees by removing optional spanning tree edges from the graph. This is performed in the following way:

1. Root nodes are selected as follows:

1.1. Every definite root node defined via the `Connections.root`-equation is a root of one spanning tree. It is an error if two (or more) definite root nodes are connected through required spanning tree edges.

1.2. 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 definite root node or one potential root node in a simulation model. If a graph of this set does not contain any definite 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.

2. If there is a cycle among required spanning-tree-edges it is an error, as it is not possible to construct a spanning tree.

3. For a subgraph with n selected roots, optional spanning-tree edges are removed such that the result is a set of n spanning trees with the selected root nodes as roots.

9.4.2.3 Equations

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

1. For every remaining optional spanning-tree edge in any of the spanning trees, the connection equations are generated according to section 9.2. For `connect(a, b)` with overdetermined connector `R`, this corresponds to the optional spanning-tree edge between `a.R` and `b.R` generating the equation `a.R = b.R`.

2. For every remaining optional spanning-tree edge 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

[*Example:*

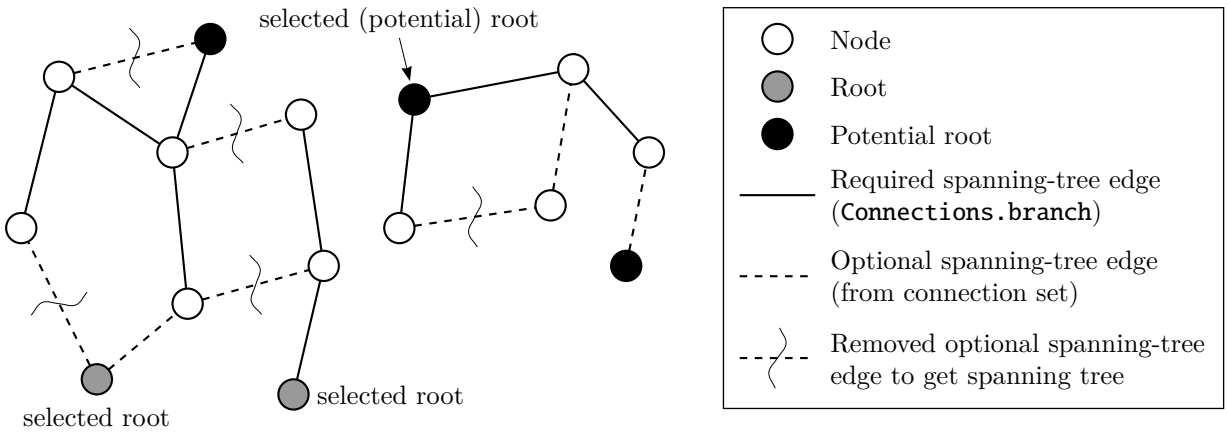


Figure 9.2: *Example of a virtual connection graph.*

]

9.4.3.1 A Power Systems Overdetermined Connector

[*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.Units.SI.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 edges are identical */
    assert(abs(theta1 - theta2) < 1.e-10, "Consistent angles");
  end equalityConstraint;
end AC_Angle;

connector AC_Plug "3-phase alternating current connector"
  import Modelica.Units.SI;
  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 theta 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;
  Real omega;
equation
  Connections.branch(p.theta,n.theta); //edge 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;
  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);
  p.theta = 2*Modelica.Constants.pi*50*time; // 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.*

Note that the different sources do not integrate the frequency, as that increases the risk of numerical errors.]

9.4.3.2 A 3-Dimensional Mechanical Systems Overdetermined Connector

[*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 TransformationMatrix;
  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 Modelica.Units.SI;
  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.Units.SI.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`*, an edge 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 of the specialized classes **type**, **record**, and **connector** can be regarded as a collection of type compatible values, section 6.7. Thus an array of the specialized classes **record** or **connector** may contain scalar values whose elements differ in their dimension sizes, but apart from that they must be of the same type. Such heterogenous arrays may only be used completely, sliced as specified, or indexed. An array of other specialized classes can only be used sliced as specified, or indexed. 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 (sing. *matrix*) 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. However, the sizes of array dimensions can be computed at run-time.

The fixed number of array dimensions permits strong type checking and efficient implementation. The non-fixed sizes of array dimensions on the other hand, allow 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, ndim=2), and arrays of more than two dimensions.

[*There is no distinction 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 `EB` for a class `EB` that is an enumeration type or subtype of the `Boolean` type.

Colon (`:`) indicates that the dimension upper bound is unknown and is a subtype of `Integer`. The size of such a variable can be determined from its binding equation, or the size of any of its array attributes, see also section 12.4.5. The size cannot be determined from other equations or algorithms.

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 `x1 = x2` 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. The notation* `EB` *stands for an enumeration type or* `Boolean`*. The general array can have one or more dimensions ($k \geq 1$).*

Modelica form 1	*Modelica form 2*	*# dims*	*Designation*	*Explanation*
`C x;`	`C x;`	0	Scalar	Scalar
`C[`n`] x;`	`C x[`n`];`	1	Vector	n-vector
`C[EB] x;`	`C x[EB]`	1	Vector	Vector indexed by `EB`
`C[`n`, `m`] x;`	`C x[`n`, `m`];`	2	Matrix	$n \times m$ matrix
`C[`n_1`, `n_2`, ..., `n_k`] x;`	`C x[`n_1`, `n_2`, ..., `n_k`];`	k	Array	General array

A component declared with array dimensions, or where the element type is an array type, is called an *array variable*. It is a component whose components are *array elements* (see below). For an array variable, the ordering of its components matters: The kth element in the sequence of components of an array variable `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.

A component contained in an array variable is called an *array element*. 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.

[*Example: 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 allows* `c`*.*

A vector `y` *indexed by enumeration values*

```
type TwoEnums = enumeration(one,two);
Real[TwoEnums] y;
```

]

Zero-valued dimensions are allowed, so: `C x[0];` declares an empty vector, and: `C x[0, 3];` an empty matrix. Some examples of array dimensions of size one are given in table 10.2.

Table 10.2: *Special cases of declaration of arrays as 1-vectors, row-vectors, or column-vectors of arrays.*

Modelica form 1	*Modelica form 2*	*# dims*	*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	(1×1)-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	$(1 \times n)$-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* `di`*-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* `di`*-dimension. Then for every such index-access an assert statement* `assert(1 <= i and i <= size(A, di), ...)` *is generated, provided this assertion cannot be checked at compile time.*

For efficiency reasons, these implicit `assert`*-statements may be optionally suppressed.*]

10.1.1 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 `Integer` values 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 12.4.5.

10.3 Built-in Array Functions

Modelica provides a number of built-in functions that are applicable to arrays.

The `promote` function listed below is utilized to define other array operators and functions.

Expression	*Description*	*Details*
`promote(`A`, `n`)`	Append dimensions of size 1	Operator 10.1

Operator 10.1 `promote`

`promote(`A`, `n`)`

Fills dimensions of size 1 from the right to array A upto dimension n, where $n \geq$ `ndims(`A`)` is required.

Let `C = promote(A, `n`)`, with $n_A =$ `ndims(A)`, then `ndims(C) = `n, `size(C, `j`) = size(A, `j`)` for $1 \leq j \leq n_A$, `size(C, `j`) = 1` for $n_A + 1 \leq j \leq n$, `C[`i_1`, ..., `i_{n_A}`, 1, ..., 1] = A[`i_1`, ..., `i_{n_A}`]`

The argument n must be a constant that can be evaluated during translation, as it determines the number of dimensions of the returned array.

[*An n that is not a constant that can be evaluated during translation for* `promote` *complicates matrix handling as it can change matrix-equations in subtle ways (e.g., changing inner products to matrix multiplication).*]

[*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 Dimension and Size Functions

The functions listed below operate on the array dimensions of the type of an expression:

Expression	*Description*	*Details*
`ndims(`A`)`	Number of dimensions	Operator 10.2
`size(`A`, `i`)`	Size of single array dimension	Operator 10.3
`size(`A`)`	Sizes of all array dimensions	Operator 10.4

Operator 10.2 `ndims`

`ndims(`A`)`

Returns the number of dimensions k of expression A, with $k \geq 0$.

Operator 10.3 `size`

`size(`A`, `i`)`

Returns the size of dimension i of array expression A where $0 \leq i \leq$ `ndims(`A`)`.

If A refers to a component of an expandable connector, then the component must be a declared component of the expandable connector, and it must not use colon (`:`) to specify the array size of dimension i.

Operator 10.4 `size`

`size(`A`)`

Returns a vector of length `ndims(`A`)` containing the dimension sizes of A.

If A refers to a component of an expandable connector, then the component must be a declared component of the expandable connector, and it must not use colon (`:`) to specify the size of any array dimension.

10.3.2 Dimensionality Conversion Functions

The conversion functions listed below convert scalars, vectors, and arrays to scalars, vectors, or matrices by adding or removing 1-sized dimensions.

Expression	*Description*	*Details*
`scalar(`A`)`	Extract only element	Operator 10.5
`vector(`A`)`	Vector of all elements	Operator 10.6
`matrix(`A`)`	Two-dimensional array	Operator 10.7

Operator 10.5 `scalar`

`scalar(`A`)`

Returns the single element of array A. `size(`A`, i)` $= 1$ is required for $1 \leq i \leq$ `ndims(`A`)`.

Operator 10.6 `vector`

`vector(`A`)`

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.

Operator 10.7 `matrix`

`matrix(`A`)`

Returns `promote(`A`, 2)` if A is a scalar or vector, and otherwise returns the elements of the first two dimensions as a matrix. `size(`A`, i)` $= 1$ is required for $2 < i \leq$ `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 listed 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.

Expression	*Description*	*Details*
`identity(`n`)`	Identity matrix	Operator 10.8
`diagonal(`v`)`	Diagonal matrix	Operator 10.9
`zeros(`n_1`, `n_2`, `n_3`, ...)`	Array with all elements being 0	Operator 10.10
`ones(`n_1`, `n_2`, `n_3`, ...)`	Array with all elements being 1	Operator 10.11
`fill(`s`, `n_1`, `n_2`, `n_3`, ...)`	Array with all elements equal	Operator 10.12
`linspace(`x_1`, `x_2`, `n`)`	Vector with equally spaced elements	Operator 10.13

Operator 10.8 `identity`

`identity(`n`)`

Returns the $n \times n$ `Integer` identity matrix, with ones on the diagonal and zeros at the other places.

Operator 10.9 `diagonal`

`diagonal(`v`)`

Returns a square matrix with the elements of vector v on the diagonal and all other elements zero.

Operator 10.10 zeros

`zeros(`n_1`, `n_2`, `n_3`, ...)`

Returns the $n_1 \times n_2 \times n_3 \times \ldots$ `Integer` array with all elements equal to zero ($n_i \geq 0$). The function needs one or more arguments, that is, `zeros()` is not legal.

Operator 10.11 ones

`ones(`n_1`, `n_2`, `n_3`, ...)`

Return the $n_1 \times n_2 \times n_3 \times \ldots$ `Integer` array with all elements equal to one ($n_i \geq 0$). The function needs one or more arguments, that is, `ones()` is not legal.

Operator 10.12 fill

`fill(`s`, `n_1`, `n_2`, `n_3`, ...)`

Returns the $n_1 \times n_2 \times n_3 \times \ldots$ 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: `fill(`s`, `n_1`, `n_2`, `n_3`, ...)` = `fill(fill(`s`, `n_2`, `n_3`, ...), `n_1`)`; `fill(`s`, `n`)` = `{`s`, `s`, ..., `s`}`.

The function needs two or more arguments; that is, `fill(`s`)` is not legal.

Operator 10.13 linspace

`linspace(`x_1`, `x_2`, `n`)`

Returns a `Real` vector with n equally spaced elements, such that `v = linspace(`x_1`, `x_2`, `n`)` results in

$$\texttt{v[}i\texttt{]} = x_1 + (x_2 - x_1)\frac{i-1}{n-1} \quad \text{for } 1 \leq i \leq n$$

It is required that $n \geq 2$. The arguments x_1 and x_2 shall be numeric scalar expressions.

10.3.4 Reduction Functions and Operators

The reduction functions listed below "reduce" an array (or several scalars) to one value (normally a scalar, but the `sum` reduction function may give an array as result and also be applied to an operator record). Note that none of these operators (particularly `min` and `max`) generate events themselves (but arguments could generate events). The restriction on the type of the input in section 10.3.4.1 for reduction expressions also applies to the array elements/scalar inputs for the reduction operator with the same name.

Expression	*Description*	*Details*
`min(`A`)`	Least element of array	Operator 10.14
`min(`x`, `y`)`	Least of two scalars	Operator 10.15
`min(... for ...)`	Reduction to least value	Operator 10.16
`max(`A`)`	Greatest element of array	Operator 10.17
`max(`x`, `y`)`	Greatest of two scalars	Operator 10.18
`max(... for ...)`	Reduction to greatest value	Operator 10.19
`sum(`A`)`	Sum of scalar array elements	Operator 10.20
`sum(... for ...)`	Sum reduction	Operator 10.21
`product(`A`)`	Product of scalar array elements	Operator 10.22
`product(... for ...)`	Product reduction	Operator 10.23

Operator 10.14 min

`min(`A`)`

Returns the least element of array expression A; as defined by <.

Operator 10.15 min

`min(`x`, `y`)`

Returns the least element of the scalars x and y; as defined by <.

Operator 10.16 min

`min(`e`(`i`, ..., `j`) for `i` in `u`, ..., `j` in `v`)`

Also described in section 10.3.4.1. Returns the least value (as defined by <) of the scalar expression e`(`i`, ..., `j`)` evaluated for all combinations of i in u, ..., j in v.

Operator 10.17 max

`max(`A`)`

Returns the greatest element of array expression A; as defined by >.

Operator 10.18 max

`max(`x`, `y`)`

Returns the greatest element of the scalars x and y; as defined by >.

Operator 10.19 max

`max(`e`(`i`, ..., `j`) for `i` in `u`, ..., `j` in `v`)`

Also described in section 10.3.4.1. Returns the greatest value (as defined by >) of the scalar expression e`(`i`, ..., `j`)` evaluated for all combinations of i in u, ..., j in v.

Operator 10.20 sum

`sum(`A`)`

Returns the scalar sum of all the elements of array expression `A`. Equivalent to sum reduction (see below, including application to operator records) over all array indices: `sum(`A`[j, k, ...] for j, k, ...)`

Operator 10.21 sum

`sum(`e`(`i`, ..., `j`) for `i` in `u`, ..., `j` in `v`)`

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.

The `sum` reduction function (both variants) may be applied to an operator record, provided that the operator record defines `'0'` and `'+'`. It is then assumed to form an additive group.

For `Integer` indexing this is

```
e(u[1], ..., v[1]) + e(u[2], ..., v[1]) + ...
 + e(u[end], ..., v[1]) + ...
 + e(u[end], ..., v[end])
```

For non-`Integer` indexing this uses all valid indices instead of 1..`end`.

The type of `sum(`e`(`i`, ..., `j`) for `i` in `u`, ..., `j` in `v`)` is the same as the type of e`(`i`, ..., `j`)`.

Operator 10.22 product

`product(`A`)`

Returns the scalar product of all the elements of array expression `A`. Equivalent to product reduction (see below) over all array indices: `product(`A`[j, k, ...] for j, k, ...)`

Operator 10.23 product

`product(`e`(`i`, ..., `j`) for `i` in `u`, ..., `j` in `v`)`

Also described in section 10.3.4.1. Returns the product of the expression e`(`i`, ..., `j`)` evaluated for all combinations of i in u, ..., j in v.

For `Integer` indexing this is

```
e(u[1], ..., v[1]) * e(u[2], ..., v[1]) * ...
 * e(u[end], ..., v[1]) * ...
 * e(u[end], ..., v[end])
```

For non-`Integer` indexing this uses all valid indices instead of 1..**end**.

The type of **product**(*e*(*i*, ..., *j*) **for** *i* **in** *u*, ..., *j* **in** *v*) 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. If **expression1** contains event-generating expressions, the expressions inside the iterators shall be evaluable.

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**-loops. 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; and for using types as ranges see section 11.2.2.2.

Table 10.3: *Reduction expressions with iterators. (The least and greatest values of* `Real` *are available as* `-Modelica.Constants.inf` *and* `Modelica.Constants.inf`*, respectively.)*

Reduction	*Restriction on* **expression1**	*Result for empty* **expression2**
sum	`Integer` or `Real`	`zeros(...)`
product	Scalar `Integer` or `Real`	1
min	Scalar enumeration, `Boolean`, `Integer` or `Real`	Greatest value of type
max	Scalar enumeration, `Boolean`, `Integer` or `Real`	Least value of type

[*Example:*

```
sum(i for i in 1:10) // Gives ∑_{i=1}^{10} i = 1 + 2 + ... + 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 ∑_{i∈{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

Functions for matrix and vector algebra are listed below. The function **transpose** can be applied to any matrix. The functions **outerProduct**, **symmetric**, **cross** and **skew** require `Real` vector(s) or matrix as input(s) and return a `Real` vector or matrix.

Expression	*Description*	*Details*
`transpose(`A`)`	Matrix transpose	Operator 10.24
`outerProduct(`x`, `y`)`	Vector outer product	Function 10.1
`symmetric(`A`)`	Symmetric matrix, keeping upper part	Function 10.2
`cross(`x`, `y`)`	Cross product	Function 10.3
`skew(`x`)`	Skew symmetric matrix associated with vector	Function 10.4

Operator 10.24 `transpose`

`transpose(`A`)`

Permutes the first two dimensions of array A. It is an error if array A does not have at least 2 dimensions.

Function 10.1 `outerProduct`

`outerProduct(`x`, `y`)`

Returns the outer product of vectors x and y, that is: `matrix(`x`) * transpose(matrix(`y`))`

Function 10.2 `symmetric`

`symmetric(`A`)`

Returns a symmetric matrix which is identical to the square matrix A on and above the diagonal.

That is, if `B := symmetric(`A`)`, then `B` is given by:

$$\texttt{B[}i,j\texttt{]} = \begin{cases} A\texttt{[}i,j\texttt{]} & \text{if } i \le j \\ A\texttt{[}j,i\texttt{]} & \text{if } i > j \end{cases}$$

Function 10.3 `cross`

`cross(`x`, `y`)`

Returns the cross product of the 3-vectors x and y:

```
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] ])
```

Function 10.4 `skew`

`skew(`x`)`

Returns the 3×3 skew symmetric matrix associated with a 3-vector, i.e., `cross(`x`, `y`) = skew(`x`) * `y. Equivalently, `skew(`x`)` is given by:

```
[ 0,    -x[3], x[2] ;
  x[3], 0,    -x[1] ;
 -x[2], x[1], 0     ]
```

10.4 Vector, Matrix and Array Constructors

The *array 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., `size(A) = size(B) = size(C) = ...`
- All arguments must be type compatible expressions (section 6.7) 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., `ndims(array(A, B, C)) = ndims(A) + 1 = ndims(B) + 1, ...`

- `{A, B, C, ...}` is a shorthand notation for `array(A, B, C, ...)`.
- There must be at least one argument.

  [*The reason* `array()` *or* `{}` *is not defined is that at least one argument is needed to determine the type of the resulting array.*]

[*Example:*

```
{1, 2, 3} is a 3-vector of type Integer.
{{11, 12, 13}, {21, 22, 23}} is a 2 x 3 matrix of type Integer
{{{1.0, 2.0, 3.0}}} is a 1 x 1 x 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 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. If `expression` contains event-generating expressions, the expressions inside the iterators shall be evaluable. They are evaluated once for each array constructor, and are 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**-loops. The loop-variable has the same type as the type of the elements of `array_expression`; and can be simple type as well as a record type. The loop-variable will have the same type for the entire loop – i.e., for an `array_expression {1, 3.2}` the iterator will have the type of the type-compatible expression (`Real`) for all iterations. For deduction of ranges, see section 11.2.2.1; and for using types as range see section 11.2.2.2.

10.4.1.1 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 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 toeplitz[:,:] = {i-j for i in 1:n, j in 1:n};
Real toeplitz2[:,:] = {{i-j for i in 1:n} for j in 1:n};
```

]

10.4.2 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., `ndims(A)` = `ndims(B)` = ...
- Arrays `A`, `B`, `C`, ... must be type compatible expressions (section 6.7) 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., $1 \leq k \leq$ `ndims(A)` = `ndims(B)` = `ndims(C)`; k shall be a parameter expression of `Integer` type.
- Size matching: Arrays `A`, `B`, `C`, ... must have identical array sizes with the exception of the size of dimension k, i.e., `size(A,` j`)` = `size(B,` j`)`, for $1 \leq j \leq$ `ndims(A)` and $j \neq k$.

[*Example:*

```
Real[2,3] r1 = cat(1, {{1.0, 2.0, 3}}, {{4, 5, 6}});
Real[2,6] r2 = cat(2, r1, 2*r1);
```

]

Formally, the concatenation `R = cat(`k`, A, B, C, ...)` is defined as follows. Let n = `ndims(A)` = `ndims(B)` = `ndims(C)` = ... Then the size of `R` is given by

```
size(R,k) = size(A,k) + size(B,k) + size(C,k) + ...
size(R,j) = size(A,j) = size(B,j) = size(C,j) = ... for 1 ≤ j ≤ n and j ≠ k
```

and the array elements of `R` are given by

```
R[i1, ..., ik, ..., in] = A[i1, ..., ik, ..., in]
  for 0 < ik ≤ size(A,k)
R[i1, ..., ik, ..., in] = B[i1, ..., ik - size(A,k), ..., in]
  for size(A,k) < ik ≤ size(A,k) + size(B,k)
R[i1, ..., ik, ..., in] = C[i1, ..., ik - size(A,k) - size(B,k), ..., in]
  for size(A,k) + size(B,k) < ik ≤ size(A,k) + size(B,k) + size(C,k)
...
```

where $1 \leq i_j \leq$ `size(R,`j`)` for $1 \leq j \leq n$.

10.4.2.1 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; ...] = cat(1, promote(A, n), promote(B, n), promote(C, n), ...)` where `n = max(2, ndims(A), ndims(B), ndims(C), ...)`. 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, ...] = cat(2, promote(A, n), promote(B, n), promote(C, n), ...)` where `n = max(2, ndims(A), ndims(B), ndims(C), ...)`. 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. `[..., ...]` has higher precedence than `[...; ...]`, e.g., `[a, b; c, d]` is parsed as `[[a, b]; [c, d]]`.
- `[A] = promote(A, max(2, 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).

[*Example:*

```
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) x 1 matrix
[M1;M2] is a (m1+m2) x 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`,` $j+1$`, ...,` k`}`, if j and k are of type `Integer`.
- j `:` k is the `Real` vector `{`j`,` $j+1.0$`, ...,` $j+n$`}`, with $n =$ `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`,` $j+d$`, ...,` $j+nd$`}`, with $n =$ `div`$(k-j,d)$, if j, d, and k are of type `Integer`.
- j `:` d `:` k is the `Real` vector `{`j`,` $j+d$`, ...,` $j+nd$`}`, with $n =$ `floor`$((k-j)/d)$, if j, d, or k are of type `Real`. In order to avoid rounding issues for the length it is recommended to use `{j + d * i for i in 0 : n}` or `linspace(j, k, n + 1)` – if the number of elements are known.
- 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 `{false, 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, ..., E.ej}` where `E.ej` > `E.ei`, and `ei` and `ej` belong to some enumeration type `E = enumeration(..., ei, ..., ej, ...)`.

[*Example:*

```
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 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). 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* (or *subscript*) expressions:

arrayname`[`*indexexpr*$_1$`,` *indexexpr*$_2$`, ...]`

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. If the number of index arguments is smaller than the number of dimensions of the array, the trailing indices will use ‘`:`’.

It is possible to index a general expression by enclosing it in parenthesis. Note that while the subscripts are applied to a ***`output-expression-list`*** in the grammar, it is only semantically valid when the ***`output-expression-list`*** represents an expression.

It is also possible to use the array access operator to assign to element/elements of an array in algorithm sections. This is called an *indexed assignment statement*. 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 are evaluated before any element is assigned a new value.

[*An indexing operation is assumed to take constant time, i.e., largely independent of the size of the array.*]

[*Example: Array indexing expressions:*

```
a[:, j]       // Vector of the j'th column of a.
a[j]          // Vector of the j'th row of a. Same as: a[j, :]
a[j : k]      // Same as: {a[j], a[j+1], ..., a[k]}
a[:, j : k]   // Same as: [a[:, j], a[:, j+1], ..., a[:, k]]
```

The range vector operator is just a special case of a vector expression:

```
v[2 : 2 : 8] // Same as: v[{2, 4, 6, 8}]
```

Array indexing in assignment statements:

```
v[{j, k}] := {2, 3}; // Same as: v[j] := 2; v[k] := 3;
v[{1, 1}] := {2, 3}; // Same as: v[1] := 3;
```

Array indexing of general expression:

```
(a*a)[:, j]     // Vector of the j'th column of a*a
```

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).*]

Table 10.4: *Examples of scalars vs. array slices created with the colon index. The examples make use of the array variables* `x[`n`,`m`]`, `v[`k`]`, *and* `z[`i`,`j`,`p`]`.

Expression	*# dims*	*Description*
`x[1, 1]`	0	Scalar
`x[:, 1]`	1	n-vector
`x[1, :]` or `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[scalar([1]), :]`	1	m-vector
`x[vector([1]), :]`	2	$1 \times m$ "row" matrix

10.5.1 Boolean or Enumeration Indices

Arrays can be indexed using values of enumeration types or the `Boolean` type, not only by `Integer`. The type of the index should correspond to the type used for declaring the dimension of the array.

[*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;
  b2[ShirtSizes.medium] := 4; // Error, b2 was declared with Boolean dimension
  w[1] := 3; // Error, w was declared with ShirtSizes dimension
```

]

10.5.2 Indexing with end

The expression **`end`** may only appear inside array subscripts, and if used in the ith subscript of an array expression `A` it is equivalent to the upper bound of the ith dimension of `A`. If used inside nested array subscripts it refers to the most closely nested array.

[*If indices to* `A` *are a subtype of* `Integer` *it is equivalent to* `size(A, i)`.]

[*Example:*

```
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)] // First end is referring to end of v.

Real B[Boolean];
B[end] is B[true]
```

]

10.6 Scalar, Vector, Matrix, and Array Operator Functions

The mathematical operations defined on scalars, vectors, and matrices are the subject of linear algebra.

The term numeric or numeric class is used below for a subtype of the `Real` or `Integer` type classes. The standard type coercion defined in section 10.6.13 applies.

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. See section 10.5 regarding assignments to array variables with vector of subscripts.

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 10.5: *Equality and assignment of arrays and scalars. The scalar Operation applies for all j in $1, \ldots, n$ and k in $1, \ldots, m$.*

Size of `a`	*Size of* `b`	*Size of* `a = b`	*Operation*
Scalar	Scalar	Scalar	`a = b`
n-vector	n-vector	n-vector	`a[`j`] = b[`j`]`
$n \times m$ matrix	$n \times m$ matrix	$n \times m$ matrix	`a[`j`, `k`] = b[`j`, `k`]`
$n \times m \times \ldots$	$n \times m \times \ldots$	$n \times m \times \ldots$	`a[`j`, `k`, ...] = b[`j`, `k`, ...]`

10.6.2 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`. Unary plus and minus are defined 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 `size(a) = size(b)`.

Table 10.6: *Array addition, subtraction, and string concatenation. In this table the symbolic operator $\pm$ represents either `+` or `-`. The scalar Operation applies for all j in $1, \ldots, n$ and k in $1, \ldots, m$.*

Size of `a`	*Size of* `b`	*Size of* `a` $\pm$ `b`	*Operation* `c := a` $\pm$ `b`
Scalar	Scalar	Scalar	`c := a` $\pm$ `b`
n-vector	n-vector	n-vector	`c[`j`] := a[`j`]` $\pm$ `b[`j`]`
$n \times m$ matrix	$n \times m$ matrix	$n \times m$ matrix	`c[`j`, `k`] := a[`j`, `k`]` $\pm$ `b[`j`, `k`]`
$n \times m \times \ldots$	$n \times m \times \ldots$	$n \times m \times \ldots$	`c[`j`, `k`, ...] := a[`j`, `k`, ...]` $\pm$ `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 10.7: *Array element-wise addition, subtraction, and string concatenation. In this table the symbolic operator $\pm$ represents either `+` or `-`, and when preceded by a dot (`.`$\pm$), either `.+` or `.-`. The scalar Operation applies for all j in $1, \ldots, n$ and k in $1, \ldots, m$.*

Size of `a`	*Size of* `b`	*Size of* `a .`$\pm$ `b`	*Operation* `c := a .`$\pm$ `b`
Scalar	Scalar	Scalar	`c := a` $\pm$ `b`
Scalar	$n \times m \times \ldots$	$n \times m \times \ldots$	`c[`j`, `k`, ...] := a` $\pm$ `b[`j`, `k`, ...]`
$n \times m \times \ldots$	Scalar	$n \times m \times \ldots$	`c[`j`, `k`, ...] := a[`j`, `k`, ...]` $\pm$ `b`
$n \times m \times \ldots$	$n \times m \times \ldots$	$n \times m \times \ldots$	`c[`j`, `k`, ...] := a[`j`, `k`, ...]` $\pm$ `b[`j`, `k`, ...]`

Table 10.8: *Unary operators. In this table the symbolic operator ± represents either unary + or unary -. The element-wise (.+, .-) and normal (+, -) operators give the same results. The scalar Operation applies for all j in $1, \ldots, n$ and k in $1, \ldots, m$.*

Size of `a`	*Size of* ± `a`	*Operation* `c :=` ± `a`
Scalar	Scalar	`c :=` ± `a`
$n \times m \times \ldots$	$n \times m \times \ldots$	`c[`j`, `k`, ...] :=` ± `a[`j`, `k`, ...]`

10.6.3 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 10.9: *Scalar and scalar to array multiplication of numeric elements. The scalar Operation applies for all j in $1, \ldots, n$ and k in $1, \ldots, m$.*

Size of `s`	*Size of* `a`	*Size of* `s * a` *and* `a * s`	*Operation* `c := s * a` *or* `c := a * s`
Scalar	Scalar	Scalar	`c := s * a`
Scalar	n-vector	n-vector	`c[`j`] := s * a[`j`]`
Scalar	$n \times m$ matrix	$n \times m$ matrix	`c[`j`, `k`] := s * a[`j`, `k`]`
Scalar	$n \times m \times \ldots$	$n \times m \times \ldots$	`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 10.10: *Array element-wise multiplication. The scalar Operation applies for all j in $1, \ldots, n$ and k in $1, \ldots, m$.*

Size of `a`	*Size of* `b`	*Size of* `a .* b`	*Operation* `c := a .* b`
Scalar	Scalar	Scalar	`c := a * b`
Scalar	$n \times m \times \ldots$	$n \times m \times \ldots$	`c[`j`, `k`, ...] := a * b[`j`, `k`, ...]`
$n \times m \times \ldots$	Scalar	$n \times m \times \ldots$	`c[`j`, `k`, ...] := a[`j`, `k`, ...] * b`
$n \times m \times \ldots$	$n \times m \times \ldots$	$n \times m \times \ldots$	`c[`j`, `k`, ...] := a[`j`, `k`, ...] * b[`j`, `k`, ...]`

10.6.4 Multiplication of Matrices and Vectors

Multiplication `a * b` of numeric vectors and matrices is defined only for the following combinations:

Table 10.11: *Matrix and vector multiplication of arrays with numeric elements. The scalar Operation applies for all i in $1, \ldots, l$ and j in $1, \ldots, n$, and the summation over k goes from 1 to m.*

Size of `a`	*Size of* `b`	*Size of* `a * b`	*Operation* `c := a * b`
m-vector	m-vector	Scalar	`c :=` $\sum_k$ `a[`k`] * b[`k`]`
m-vector	$m \times n$ matrix	n-vector	`c[`j`] :=` $\sum_k$ `a[`k`] * b[`k`, `j`]`
$l \times m$ matrix	m-vector	l-vector	`c[`i`] :=` $\sum_k$ `a[`i`, `k`] * b[`k`]`
$l \times m$ matrix	$m \times n$ matrix	$l \times n$ matrix	`c[`i`, `j`] :=` $\sum_k$ `a[`i`, `k`] * b[`k`, `j`]`

[*Example:*

```
Real A[3, 3], x[3], b[3], v[3];
A * x = b;
x * A = b; // same as transpose([x])*A*b
[v] * transpose([v]) // outer product
v * A * v // scalar
transpose([v]) * A * v // vector with one element
```

]

10.6.5 Division 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 10.12: *Division of scalars and arrays by numeric elements. The scalar Operation applies for all j in $1, \ldots, n$ and k in $1, \ldots, m$.*

Size of `a`	*Size of* `s`	*Size of* `a / s`	*Operation* `c := a / s`
Scalar	Scalar	Scalar	`c := a / s`
n-vector	Scalar	n-vector	`c[`k`] := a[`k`] / s`
$n \times m$ matrix	Scalar	$n \times m$ matrix	`c[`j`, `k`] := a[`j`, `k`] / s`
$n \times m \times \ldots$	Scalar	$n \times m \times \ldots$	`c[`j`, `k`, ...] := a[`j`, `k`, ...] / s`

10.6.6 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 `size(a) = size(b)` or scalar `a` or scalar `b`. The result is always of `Real` type. In order to get integer division with truncation, use the function `div`.

Table 10.13: *Element-wise division of arrays. The scalar Operation applies for all j in $1, \ldots, n$ and k in $1, \ldots, m$.*

Size of `a`	*Size of* `b`	*Size of* `a ./ b`	*Operation* `c := a ./ b`
Scalar	Scalar	Scalar	`c := a / b`
Scalar	$n \times m \times \ldots$	$n \times m \times \ldots$	`c[`j`, `k`, ...] := a / b[`j`, `k`, ...]`
$n \times m \times \ldots$	Scalar	$n \times m \times \ldots$	`c[`j`, `k`, ...] := a[`j`, `k`, ...] / b`
$n \times m \times \ldots$	$n \times m \times \ldots$	$n \times m \times \ldots$	`c[`j`, `k`, ...] := a[`j`, `k`, ...] / b[`j`, `k`, ...]`

[*Example: Element-wise division by scalar (`./`) and division by scalar (`/`) are identical:* `a ./ s = a / s`*:*

```
2./[1, 2; 3, 4]  // error; same as  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 Element-wise Exponentiation

Exponentiation `a ^ b` always returns a `Real` scalar value, and it is required that `a` and `b` are scalar `Real` or `Integer` expressions. The result should correspond to mathematical exponentiation with the following special cases:

- For any value of `a` (including 0.0) and an `Integer` `b` $= 0$, the result is 1.0.
- If `a` < 0 and `b` is an `Integer`, the result is defined as $\pm|a|^b$, with sign depending on whether `b` is even (positive) or odd (negative).
- A deprecated semantics is to treat `a` < 0 and a `Real` `b` having a non-zero integer value as if `b` were an `Integer`.
- For `a` $= 0$ and `b` > 0, the result is 0.0.
- Other exceptional situations are illegal. For example: `a` $= 0.0$ and `b` $= 0.0$ for a `Real` `b`, `a` $= 0.0$ and `b` < 0, or `a` < 0 and `b` does not have an integer value.

[*Except for defining the special case of* 0.0^0 *it corresponds to* `pow(double a, double b)` *in the ANSI C library. The result is always* `Real` *as negative exponents can give non-integer results also when both operands are* `Integer`*. The special treatment of* `Integer` *exponents makes it possible to use* x^n *in a power series.*]

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 `size(a) = size(b)` or scalar `a` or scalar `b`.

Table 10.14: *Element-wise exponentiation of arrays. The scalar Operation applies for all j in $1, \ldots, n$ and k in $1, \ldots, m$.*

Size of `a`	*Size of* `b`	*Size of* `a .^ b`	*Operation* `c := a .^ b`
Scalar	Scalar	Scalar	`c := a ^ b`
Scalar	$n \times m \times \ldots$	$n \times m \times \ldots$	`c[`j`, `k`, ...] := a ^ b[`j`, `k`, ...]`
$n \times m \times \ldots$	Scalar	$n \times m \times \ldots$	`c[`j`, `k`, ...] := a[`j`, `k`, ...] ^ b`
$n \times m \times \ldots$	$n \times m \times \ldots$	$n \times m \times \ldots$	`c[`j`, `k`, ...] := a[`j`, `k`, ...] ^ b[`j`, `k`, ...]`

[*Example:*

```
2.^[1, 2; 3, 4]  // error; same as 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 Matrices

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 computing 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, ...}`.
- If `m` is also an array component, the slice operation is valid only if `size(a[1].m) = size(a[2].m) = ...`
- 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], ...}`, and does not require that `size(a[1].m) = size(a[2].m)`. The number of subscripts on `m` must not be greater than the number of array dimension for `m` (the number can be smaller, in which case the missing trailing indices are assumed to be '`:`'), and is only valid if `size(a[1].m[...]) = size(a[2].m[...])`.

[*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[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 12.4.6.

10.6.13 Standard Type Coercion

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**.

This also applies to arrays of **Real**, and for fields of record expressions. There is no similar rule for sub-typing.

[*Example:*

```
record RealR
  Real x,y;
end RealR;
record IntegerR
  Integer x,y;
end IntegerR;
parameter Integer a = 1;
Real y(start=a);           // Ok, a is automatically coerced to Real
RealR r1 = IntegerR(a, a); // Ok, record is automatically coerced
RealR r2 = RealR(a, a);    // Ok, a is automatically coerced to Real
```

]

10.7 Empty Arrays

Arrays may have dimension sizes of 0. For example:

```
Real x[0]; // an empty vector
Real A[0, 3], B[5, 0], C[0, 0]; // empty matrices
```

Empty matrices can be constructed using the **fill** function. For example:

```
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
```

[*Example: Whereas scalar indexing into an empty dimension of an array is an error, not all applications of indices to empty arrays are invalid:*

```
Real[1, 0] a = fill(0.0, 1, 0); // a Real 1 x 0 matrix
Real[0] a1a = a[1]; // empty vector
Real[0] a1b = a[1, :]; // same as above
```

```
Real[0] a1c = a[1, 1 : end]; // same as above, as 1 : end is empty
```

]

Size-requirements of operations, such as +, -, must also be fulfilled if a dimension is zero. For example:

```
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) // matrix of zeros
```

Note that `fill(0.0, m, n)` will be an empty matrix if `m` or `n` is zero.

[*Example:*

```
  Real u[p], x[n], y[q], A[n, n], B[n, p], C[q, n], D[q, p];
equation
  der(x) = A * x + B * u
  y = C * x + D * u
```

Assume $\mathtt{n} = 0$, $\mathtt{p} > 0$, $\mathtt{q} > 0$*: 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

An *algorithm section* is a part of a class definition comprised of the keyword **algorithm** followed by a sequence of statements. The formal syntax is as follows:

```
algorithm-section :
   [ initial ] algorithm { statement ";" }
```

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 tool 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.

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.

11.1.2 An Algorithm in a Model

An algorithm section is conceptually a code fragment that remains together and the statements of an algorithm section are executed in the order of appearance. Whenever an algorithm section is invoked, all variables appearing on the left hand side of the assignment operator `:=` are initialized (at least conceptually):

- A continuous-time variable is initialized with the value of its `start`-attribute.
- A discrete-time variable `v` is initialized with `pre(v)`.
- If at least one element of an array appears on the left hand side of the assignment operator, then the complete array is initialized in this algorithm section.
- A parameter assigned in an initial algorithm, section 8.6, is initialized with the value of its `start`-attribute.

[*Initialization is performed, in order that an algorithm section cannot introduce a "memory" (except in the case of discrete-time variables assigned in the algorithm), which could invalidate the assumptions of*

a numerical integration algorithm. Note, a Modelica tool may change the evaluation of an algorithm section, provided the result is identical to the case, as if the above conceptual processing is performed.

An algorithm section is treated as an atomic vector-equation, which is sorted together with all other equations. For the sorting process (BLT), every algorithm section with N different left-hand side variables, is treated as an atomic N-dimensional vector-equation containing all variables appearing in the algorithm section. This guarantees that all N equations end up in an algebraic loop and the statements of the algorithm section remain together.

Example:

```
model Test // wrong Modelica model (has 4 equations for 2 unknowns)
  Real x[2](start = {-11, -22});
algorithm // conceptually: x = {1, -22}
  x[1] := 1;
algorithm // conceptually: x = {-11, 2}
  x[2] := 2;
end Test;
```

The conceptual part indicate that if the variable is assigned unconditionally in the algorithm before it is used the initialization can be omitted. This is usually the case, except for algorithms with **when***-statements, and especially for initial algorithms.*]

11.1.3 The Algorithm in a Function

See section 12.4.4 *Initialization and Binding Equations.*

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
   )
   description
```

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*.

The **expression** must not have higher variability than the assigned component, see section 3.8.

Assignment to array variables with subscripts is described in section 10.5.

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, and the variables are assigned from left to right.

```
(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[1]) := f(3, 4);
```

In the second example above `x[1]` *is assigned twice: first with the first output, and then with the third output. For that case the following will give the same result:*

```
(, x[2], x[1]) := 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.*]

11.2.1.2 Assigned Variables - Restrictions

Only components of the specialized classes **type**, **record**, **operator record**, and **connector** may appear as left-hand-side in algorithms. This applies both to simple assignment statements, and the parenthesized, comma-separated list of variables for functions with multiple results.

11.2.2 For-Statement

The syntax of a **for**-statement is as follows:

```
for for-indices loop
  { statement ";" }
end for
```

A **for**-statement may optionally use several iterators (**for**-indices), see section 11.2.2.3 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 rules for **for**-statements are the same as for **for**-expressions in section 8.3.2.1 – except that the **expression** of a **for**-statement is not restricted to a parameter-expression.

If the **for**-statement contains event-generating expressions, any expression in *for-index* shall be evaluable.

[*In general, the same event-generating expression requires distinct crossing functions for different iterations of the* **for***-loop, and the restriction ensures that the number of crossing functions is known during translation time.*]

[*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, where the expressions are not part of an array in a component of an expandable connector. 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. There may not be assignments to the entire arrays that are subscripted with `IDENT` inside the loop, but there may be assignments to individual elements or ranges of elements.

[*The size of an array – the iteration range – can be evaluated on entry to the* **for***-loop, since the array size cannot change during the execution of the* **for***-loop.*]

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. The `IDENT` may also be used as a subscript for an array in a component of an expandable connector but it is only seen as a reference to the variable `IDENT` and cannot be used for deducing ranges.

[*Example: Implicit iterator ranges for an* `Integer` *subscript:*

```
  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)];
  Real xsquared3[size(x, 1)];
equation
  for i loop // Same as: for i in 1 : size(x, 1) loop ...
    xsquared2[i] = x[i]^2;
  end for;
algorithm
  for i loop // Same as: for i in 1 : size(x, 1) loop ...
    xsquared3[i] := x[i]^2;
  end for;
```

]

[*Example: An array dimension's type of subscript does not matter for array compatibility, only the size of the array dimension matters. This is true also for array constructor expressions with implicit iterator ranges:*

```
type FourEnums = enumeration(one, two, three, four);
Real x[4];
Real xe[FourEnums] = x;
Real xsquared3[FourEnums] = {xe[i] * xe[i] for i in FourEnums};
Real xsquared4[FourEnums] = {xe[i] * xe[i] for i};
Real xsquared5[FourEnums] = {x[i] * x[i] for i};
```

]

11.2.2.2 Types as Iteration Ranges

The iteration range can be specified as `Boolean` or as an enumeration type. This means iteration over the type from min to max, i.e., for `Boolean` it is the same as `false:true` and for an enumeration `E` it is the same as `E.min:E.max`. This can be used for **for**-loops and reduction expressions.

[*Example:*

```
  type FourEnums = enumeration(one, two, three, four);
  Real xe[FourEnums];
  Real xsquared1[FourEnums];
  Real xsquared2[FourEnums] = {xe[i] * xe[i] for i in FourEnums};
equation
  for i in FourEnums loop
    xsquared1[i] = xe[i]^2;
  end for;
```

]

11.2.2.3 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 "`functionName(`".

[*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 other 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 11.2.5, is executed), and then execution proceeds at step 1.

Event-generating expressions are neither allowed in the **expression** nor in the loop body statements. A deprecated feature is that all expressions in a **while**-statement are implicitly inside noEvent.

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 that 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

The **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 ";" }
{ elsewhen expression then
  { statement ";" }
}
end when
```

The **expression** of a **when**-statement shall be a discrete-time Boolean scalar or vector expression. The statements within a **when**-statement are activated only at the instant 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 on the positive edge of any of the expressions* `x > 2`, `sample(0, 2)`, *or* `x < 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 Where a When-Statement May Occur

- A **when**-statement shall not be used within a function.
- A **when**-statement shall not occur inside an initial algorithm.
- A **when**-statement cannot be nested inside another **when**-statement.
- **when**-statements shall not occur inside **while**-loops, **for**-loops, or **if**-statements 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 Statements within When-Statements

[*In contrast to* **when**-*equations, section 8.3.5.3, there are no additional restrictions within* **when**-*statements:*

- *In algorithms, all assignment statements are already restricted to left-hand-side variables.*
- *If at least one element of an array appears on the left-hand-side of the assignment operator inside a* **when**-*statement, it is as if the entire array appears in the left-hand-side according to section 11.1.2. Thus, there is no need to restrict the indices to parameter-expressions.*
- *The* **for**-*loops and* **if**-*statements are not problematic inside* **when**-*statements in algorithms, since all left-hand-side variables inside* **when**-*statements are assigned to their pre-values before the start of the algorithm, according to section 11.1.2.*

]

11.2.7.3 Defining When-Statements by If-Statements

A **when**-statement:

```
algorithm
  when {x > 1, ..., y > p} then
    ...
  elsewhen x > y.start then
    ...
  end when;
```

is similar to the following special **if**-statement, where `Boolean b1[N];` and `Boolean b2;` are necessary because `edge` 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 ... 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 Assert-Statement

See section 8.3.7. A failed `assert` stops the execution of the current algorithm.

11.2.8.2 Terminate-Statement

See section 8.3.8. The `terminate`-statement shall not be used 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 12.9). 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 = defaultExpr_1 "Comment" annotation(...);
  ...
  output TypeO1 out1;
  output TypeO2 out2 = defaultExpr_2;
  ...
protected
  ⟨local variables⟩
  ...
algorithm
  ⟨statements⟩
  ...
end functionname;
```

]

Optional explicit default values can be associated with any input or output formal parameter through binding equations. Comment strings and annotations can be given for any formal parameter declaration, as usual in Modelica declarations.

[*Explicit default values are shown for the third input parameter and the second output parameter in the example above.*]

[*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-Statements

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 that 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 if;
  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.*]

A special case is defining a **`function`** as a short-class definition with modifiers for inputs inside a model. These default values, unless overridden in the function call, will then be considered for variability similarly as if they were given in the function call, see section 3.8.1.

[*Example: Demonstrating the variability implications. Note that functions cannot directly use non-constants in enclosing scopes, so we cannot write* `input Real x1 = x;` *directly in* `foo`.

```
model M
  function foo
    input Real x1;
    input Real x2 = 2;
    output Real y;
  algorithm
    y := x1 + x2;
  end foo;
  Real x = time;
  function f1 = foo(x1 = x);
  constant Real z1 = f1(x1 = 2); // Legal, since 'x1' has a new value.
  constant Real z2 = f1(x2 = 1); // Illegal, since 'x' is seen as an argument.
end M;
```

]

12.2 Function as a Specialized Class

The function concept in Modelica is a specialized class (section 4.7).

[*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 public component must have the prefix `input` or `output`.
- Input formal parameters are read-only after being bound to the actual arguments or default values, i.e., they shall not be assigned values in the body of the function.
- A function shall *not be used in connections*, shall not have *equations*, shall not have *initial algorithms*.
- A function can have at most *one algorithm* section or *one external function interface* (not both), which, if present, is the body of the function.
- A function may only contain components of the specialized classes **type**, **record**, **operator record**, and **function**; and it must not contain, e.g., **model**, **block**, **operator** or **connector** components.
- A function may not contain components of type `Clock`.
- The elements of a function shall not have prefixes **inner**, or **outer**.
- 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, the function shall not be partial, and the output variables must be assigned inside the function either in binding equations or in an algorithm section, or have an external function interface as its body, or be defined as a function partial derivative. The output variables of a function should be computed.

  [*It is a quality of implementation how much analysis a tool performs in order to determine if the output variables are computed.*]

  A function *cannot contain* calls to the Modelica *built-in operators* `der`, **initial**, `terminal`, `sample`, `pre`, `edge`, `change`, `reinit`, `delay`, `cardinality`, `inStream`, `actualStream`, to the operators of the built-in package `Connections`, to the operators defined in chapter 16 and chapter 17, and is not allowed to contain **when**-statements.
- The dimension *sizes* not declared with colon (`:`) of each array result or array local variable (i.e., a non-input component) 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 12.4.4).
- For initialization of local variables of a function see section 12.4.4).

- 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**:

- Functions can be called, section 12.4.
  - The calls can use a mix of positional and named arguments, see section 12.4.1.
  - Instances of functions have a special meaning, see section 12.4.2.
  - The lookup of the **function** class to be called is extended, see section 5.3.2.
- A function can be *recursive*.
- A formal parameter or local variable may be initialized through a *binding* (=) of a default value in its declaration, see section 12.4.4. Using assignment (:=) is deprecated. If a non-input component in the function uses a record class that contain one or more binding equations they are viewed as initialization of those component of the record component.
- 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 colon (`:`) to be resized for non-input array variables, see section 12.4.5.
- A function may be defined in a short function definition to be a function partial derivative.

12.3 Pure Modelica Functions

Modelica functions are normally *pure* which makes it easy for humans to reason about the code since they behave as mathematical functions, and possible for compilers to optimize.

- *Pure* Modelica functions always give the same output values or errors for the same input values and only the output values influence the simulation result, i.e., is seen as equivalent to a mathematical map from input values to output values. Some input values may map to errors. Pure functions are thus allowed to fail by calling `assert`, or `ModelicaError` in C code, or dividing by zero. Such errors will only be reported when and if the function is called. *Pure* Modelica functions are not assumed to be thread-safe.
- A Modelica function which does not have the *pure* function properties is *impure*.

The declaration of functions follow these rules:

- Functions defined in Modelica (non-external) are *normally* assumed to be pure (the exception is the deprecated case below), if they are impure they shall be marked with the **impure** keyword. They can be explicitly marked as **pure**.

  [*However, since functions as default are pure it is not recommended to explicitly declare them as* **pure**.]
- External functions must be explicitly declared with **pure** or **impure**.
- If a function is declared as **impure** any function extending from it shall be declared as **impure**.
- A deprecated semantics is that external functions (and functions defined in Modelica directly or indirectly calling them) without **pure** or **impure** keyword are assumed to be impure, but without any restriction on calling them. Except for the function `Modelica.Utilities.Streams.print`, a diagnostic must be given if called in a simulation model.

Calls of pure functions used inside expression may be skipped if the resulting expression will not depend on the possible returned value; ignoring the possibility of the function generating an error.

A call to a function with no declared outputs is assumed to have desired side-effects or assertion checks.

[*A tool shall thus not remove such function calls, with exception of non-triggered assert calls. A pure function, used in an expression or used with a non-empty left hand side, need not be called if the output from the function call do not mathematically influence the simulation result, even if errors would be generated if it were called.*]

[*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. For example, a tool may use common subexpression elimination to call a pure function just once, if it is called several times with identical input arguments. However, since functions may fail we can, e.g., only move a common function call from inside a loop to outside the loop if the loop is run at least once.*]

[*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.*]

With the prefix keyword **impure** it is stated that a Modelica function is *impure* and it is only allowed to call such a function from within:

- Another function marked with the prefix **impure**.
- A **when**-equation.
- A **when**-statement.
- **pure**(impureFunction(...)) – which allows calling impure functions in any pure context. The wrapping in **pure**(...) only by-passes the purity checking of the callee impureFunction; the argument expressions of the function call are not affected.
- Initial equations and initial algorithms.
- Binding equations for components declared as parameter – which is seen as syntactic sugar for having a parameter with fixed=false and the binding as an initial equation.

  [*Thus, evaluation of the same function call at a later time during simulation is not guaranteed to result in the same value as when the parameter was initialized, seemingly breaking the declaration equation.*]
- Binding equations for external objects.

For initial equations, initial algorithms, and bindings it is an error if the function calls are part of systems of equations and thus have to be called multiple times.

[*A tool is not allowed to perform any optimizations on function calls to an impure function, e.g., reordering calls from different statements in an algorithm or common subexpression elimination is not allowed.*]

By section 6.6, it follows that an impure function can only be passed as argument to a function formal parameter of impure type. A function having a formal function parameter that is **impure** must be marked **pure** or **impure**.

[*Comment: The semantics are undefined if the function call of an impure function is part of an algebraic loop.*]

[*Example:*

```
function evaluateLinear // pure function
  input Real a0;
  input Real a1;
  input Real x;
  output Real y;
algorithm
  y := a0 + a1*x;
end evaluateLinear;

impure function receiveRealSignal // impure function
```

```
  input HardwareDriverID id;
  output Real y;
external "C"
  y = receiveSignal(id);
end receiveRealSignal;
```

Examples of allowed optimizations of pure functions:

```
model M // Assume sin, cos, asin are pure functions with normal derivatives.
  input Real x[2];
  input Real w;
  Real y[2] = [cos(w), sin(w); -sin(w), cos(w)] * x;
  Real z[2] = der(y);
  Real a = 0 * asin(w);
end M;
```

A tool only needs to generate one call of the pure function `cos(w)` *in the model* `M` *– a single call used for both the two elements of the matrix, as well as for the derivative of that matrix. A tool may also skip the possible error for* `asin(w)` *and assume that* `a` *is zero.*

Examples of restrictions on optimizing pure functions:

```
  Real x =
    if noEvent(abs(x)) < 1 then
      asin(x) // Cannot move asin(x) out of if-branch.
    else
      0;
algorithm
  assertCheck(p, T); // Must call function
algorithm
  if b then
    y := 2 * someOtherFunction(x);
  end if;
  y := y + asin(x);
  y := y + someOtherFunction(x);
  // Cannot evaluate someOtherFunction(x) before asin(x) - unless b is true
  // The reason is that asin(x) may fail and someOtherFunction may hang,
  // and it might be possible to recover from this error.
```

]

12.4 Function Call

Function classes and record constructors (section 12.6) and enumeration type conversions (section 4.9.5.3) can be called as described in this section.

12.4.1 Positional or Named Input Arguments

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 (simplified by removing reduction expression, section 10.3.4.1):

```
primary :
   component-reference function-call-args

function-call-args :
   "(" [ function-arguments ] ")"

function-arguments :
   function-argument [ "," function-arguments]
   | named-arguments
```

```
named-arguments: named-argument [ "," named-arguments ]

named-argument: IDENT "=" function-argument

function-argument : function-partial-application | 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. The value of the argument is placed in the slot, filling it (it is an error if this slot is already filled). When all arguments have been processed, the slots that are still unfilled are filled with the corresponding default value of the function definition. The default values may depend on other inputs (these dependencies must be acyclical in the function) – the values for those other inputs will then be substituted into the default values (this process may be repeated if the default value for that input depend on another input). The default values for inputs shall not depend on non-input variables in the function. The list of filled slots is used as the argument list for the call (it is an error if any unfilled slots still remain).

Special purpose operators with function syntax defined in the specification shall not be called with named arguments, unless otherwise noted.

The type of each argument must agree with the type of the corresponding parameter, except where the standard type coercion, section 10.6.13, can be used to make the types agree. (See also section 12.4.6 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 Functional Input Arguments

A functional input argument to a function is an argument of function type. The declared type of such an input formal parameter in a function can be the type-specifier of a partial function that has no replaceable elements. It cannot be the type-specifier of a record or enumeration (i.e., record constructor functions and enumeration type conversions are not allowed in this context). Such an input formal parameter of function type can also have an optional functional default value.

[*Example:*

```
function quadrature "Integrate function y = integrand(x) from x1 to x2"
  input Real x1;
  input Real x2;
  input Integrand integrand; // Integrand is a partial function, see below
  // With default: input Integrand integrand = Modelica.Math.sin;
  output Real integral;
```

```
algorithm
  integral := (x2 - x1) * (integrand(x1) + integrand(x2)) / 2;
end quadrature;

partial function Integrand
  input Real u;
  output Real y;
end Integrand;
```

]

A functional argument can be provided in one of the following forms to be passed to a scalar formal parameter of function type in a function call:

1. as a function type-specifier (`Parabola` example below),
2. as a function partial application (section 12.4.2.1 below),
3. as a function that is a component (i.e., a formal parameter of function type of the enclosing function),
4. as a function partial application of a function that is a component (example in section 12.4.2.1 below).

In all cases the provided function must be function-compatible (definition 6.8) with the corresponding formal parameter of function type.

[*Example: A function as a positional input argument according to case 1:*

```
function Parabola
  extends Integrand;
algorithm
  y := x * x;
end Parabola;
area = quadrature(0, 1, Parabola);
```

The `quadrature2` *example below uses a function* `integrand` *that is a component as input argument according to case 3:*

```
function quadrature2 "Integrate function y = integrand(x) from x1 to x2"
  input Real x1;
  input Real x2;
  input Integrand integrand; // Integrand is a partial function type
  output Real integral;
algorithm
  integral :=
    quadrature(x1, (x1 + x2) / 2, integrand) +
    quadrature((x1 + x2) / 2, x2, integrand);
end quadrature2;
```

]

12.4.2.1 Function Partial Application

A function partial application is similar to a function call with certain formal parameters bound to expressions, the specific rules are specified in this section and are not identical to the ones for function call in section 12.4.1. A function partial application returns a partially evaluated function that is also a function, with the remaining not bound formal parameters still present in the same order as in the original function declaration. A function partial application is specified by the **`function`** keyword followed by a function call to `func_name` giving named formal parameter associations for the formal parameters to be bound, e.g.:

```
function func_name(..., formal_parameter_name = expr, ...)
```

[*Note that the keyword* **`function`** *in a function partial application differentiates the syntax from a normal function call where some parameters have been left out, and instead supplied via default values.*]

The function created by the function partial application acts as the original function but with the bound formal input parameters(s) removed, i.e., they cannot be supplied arguments at function call. The binding occurs when the partially evaluated function is created. A partially evaluated function is function-compatible (definition 6.8) with the same function where all bound arguments are removed.

[*Thus, for checking function type compatibility, bound formal parameters are ignored.*]

[*Example: Function partial application as argument, positional argument passing, according to case 2 above:*

```
model Test
  parameter Integer N;
  Real area;
algorithm
  area := 0;
  for i in 1:N loop
    area := area + quadrature(0, 1, function Sine(A = 2, w = i * time));
  end for;
end Test;

function Sine "y = Sine(x, A, w)"
  extends Integrand;
  input Real A;
  input Real w;
algorithm
  y := A * Modelica.Math.sin(w * x);
end Sine;
```

Call with function partial application as named input argument:

```
area :=
  area + quadrature(0, 1, integrand = function Sine(A = 2, w = i * time));
```

]

[*Example: Function types are matching after removing the bound arguments* `A` *and* `w` *in a function partial application:*

```
function Sine2 "y = Sine2(A, w, x)"
  input Real A;
  input Real w;
  input Real x; // Note: x is now last in argument list.
  output Real y;
algorithm
  y := A * Modelica.Math.sin(w * x);
end Sine2;
area = quadrature(0, 1, integrand = function Sine2(A = 2, w = 3));
```

The partially evaluated `Sine2` *has only one argument:* `x` *– and is thus type compatible with* `Integrand`.]

[*Example: Function partial application of a function that is a component, according to case 4 above:*

```
partial function SurfaceIntegrand
  input Real x;
  input Real y;
  output Real z;
end SurfaceIntegrand;

function quadratureOnce
  input Real x;
  input Real y1;
  input Real y2;
  input SurfaceIntegrand integrand;
  output Real z;
algorithm
  z := quadrature(y1, y2, function integrand(y = x));
```

```
  // This is according to case 4 and needs to bind the 2nd argument
end quadratureOnce;

function surfaceQuadrature
  input Real x1;
  input Real x2;
  input Real y1;
  input Real y2;
  input SurfaceIntegrand integrand;
  output Real integral;
algorithm
  integral :=
    quadrature(x1, x2,
      function quadratureOnce(y1 = y1, y2 = y2, integrand = integrand));
  // Case 2 and 3
end surfaceQuadrature;
```

]

12.4.3 Output Formal Parameters

A function may have more than one output component, corresponding to multiple return values. The only way to use more than the first return value of such a function is to make the function call the right hand side of an equation or assignment. In this case, 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 at least 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"
  input Real A[:, size(A,1)];
  output Real eigenValues[size(A,1),2];
  output Real rightEigenVectors[size(A,1),size(A,1)];
  output Real leftEigenVectors [size(A,1),size(A,1)];
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.
```

]

12.4.4 Initialization and Binding Equations

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 the function do not have **start**-attributes. However, a binding equation (**= expression**) with an expression may be present for a component.

[Declaration assignments *of the form* **:= expression** *are deprecated, but otherwise identical to binding equations.*]

A binding equation 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). These bindings must be executed in an order where a variable is not used before its binding equations has been executed; it is an error if no such order exists (i.e., the binding must be acyclic).

Binding equations can only be used for components of a function. If no binding equation is given for a non-input component the variable is uninitialized (except for record components where modifiers may also initialize that component). It is an error to use (or return) an uninitialized variable in a function. Binding equations for input formal parameters are interpreted as default arguments, as described in section 12.4.1.

[*It is recommended to check for use of uninitialized variables statically – if this is not possible a warning is recommended combined with a run-time check.*]

[*The properties of components in functions described in this section are also briefly described in section 12.2.*]

12.4.5 Flexible Array Sizes and Resizing of Arrays

[*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;
```

]

A non-input array component declared in a function with a dimension size specified by colon (`:`) and no binding equation, 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;
```

]

12.4.6 Automatic Vectorization

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 6.3.1 and section 7.1.4) may be called vectorized.

Consider the expression `f(arg1, ..., argn)`, an application of the function `f` to the arguments `arg1`, ..., `argn`. Potential vectorization of this call is defined as follows. 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 `e` with the same dimension sizes as the foreach arguments. Each element `e[`i`, ..., `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`, ..., `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.

[*Example:*

```
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
atan2({a, b, c}, {d, e, f}) = {atan2(a, d), atan2(b, e), atan2(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* `Real` *vector 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]`*. For built-in operators one can do this with* `1 .+ [1,2,3]` *but not with* `1 + [1,2,3]`*, because the rules for the built-in operators are more restrictive.*]

12.4.7 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, but it is useful to check assertions and in certain cases for desired side-effects, see section 12.3.*]

An empty call can occur either as a kind of "null equation" or "null statement".

[*Example: The empty calls to* `eigen()` *are examples of a "null equation" and a "null statement":*

```
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 3.7.4.
- Event-related operators with function syntax, see section 3.7.5.
- Built-in array functions, see section 10.3.

  Note that when the specification references a function having the name of a built-in function it references the built-in function, not a user-defined function having the same name.

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.

[*The partial flattening is performed in order to remove potentially conflicting* `import`*-clauses 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 `final parameter Real`) are declared as protected components in the record constructor function.
- Prefixes (`constant`, `parameter`, `final`, `discrete`, ...) 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 constructing 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.re);
  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;
    final 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 (the name of the output, given in italic below, is not defined; it should be chosen to not cause any conflict):

```
package Demo
  function Record1
    input Real r0 = 0;
    output Record1 result(r0 = r0);
  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
    final 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.]

12.6.1 Casting to Record

A constructor of a record `R` can be used to cast an instance m of a **model**, **block**, **connector** class `M` to a value of type `R`, provided that for each component defined in `R` (that do not have a default value) there is also a public component defined in `M` with identical name and type. A nested record component of `R` is handled as follows, if the corresponding component of `M` is a **model**/**block**/**connector** a nested record constructor is called – otherwise the component is used directly; and the resulting call/component is used as argument to the record constructor `R`. If the corresponding component of `R` in `M` is a conditional component, it is an error. The instance `m` is given as single (un-named) argument to the record constructor of `R`. The interpretation is that `R(m)` is replaced by a record constructor of type `R` where all public components of `M` that are present in `R` are assigned to the corresponding components of `R`. The record cast can be used in vectorized form according to section 12.4.6.

[*The problem if* `R` *would be a conditional component is that the corresponding binding would be illegal since it is not a* `connect`*-equation.*]

[*The record cast operation is uniquely distinguished from a record constructor call, because an argument of the record constructor cannot be a* **model**, **block** *or* **connector** *instance.*]

[*Example:*

```
connector Flange
  Real phi;
  flow Real tau;
end Flange;

model Model1
  Real m1;
  Boolean b1;
  Flange flange;
end Model1;

model Model2
  Real r1;
  Real r2;
  Integer i2
  Pin p1, p2;
  Model1 sub1;
  protected
  Integer i1;
  ...
end Model2;

record MyFlange
  Real tau;
end MyFlange;

record MyRecord1
  Boolean b1;
  MyFlange flange;
end MyRecord1;
```

```
record MyRecord2
  Real r1;
  Integer i2;
  MyRecord1 sub1;
end MyRecord2;

model Model
  Model2 s1;
  Model2 s2[2];
  MyRecord2 rec1 = MyRecord2(s1);
  MyRecord2 rec2[2] = MyRecord2(s2);
  ...
end Model;
// Model is conceptually mapped to
model ModelExpanded
  Model2 s1;
  Model2 s2[2];
  MyRecord2 rec1 = MyRecord2(r1=s1.r1, i2=s1.i2,
  sub1 = MyRecord1(b1=s1.sub1.b1,
  flange = MyFlange(tau=s1.sub1.flange.tau));
  MyRecord2 rec2[2] = {MyRecord2(r1=s2[1].r1, i2=s2[1].i2,
  sub1 = MyRecord1(b1=s2[1].sub1.b1,
  flange = MyFlange(tau=s1[1].sub1.flange.tau)),
  MyRecord2(r1=s2[2].r1, i2=s2[2].i2,
  sub1 = MyRecord1(b1=s2[2].sub1.b1,
  flange = MyFlange(tau=s2[2].sub1.flange.tau)};
  ...
end ModelExpanded;
```

]

12.7 Derivatives and Inverses of Functions

The annotations listed below are related to differentiation and closed-form inverses of functions. A function declaration can have `derivative` annotations specifying derivative functions or preferably, for a function written in Modelica, use the `smoothOrder` annotation to indicate that the tool can construct the derivative function automatically. Partial derivatives are not provided via annotations, but using a certain type of short function definition described in section 12.7.2.

Annotation	*Description*	*Details*
`smoothOrder`	Function smoothness guarantee	Annotation 12.1
`derivative`	Provide function derivative	Annotation 12.2
`inverse`	Provide closed-form inverses	Annotation 12.3

Annotation 12.1 `smoothOrder`

```
"smoothOrder" "=" UNSIGNED-NUMBER
"smoothOrder"
   "("
      "normallyConstant" "=" IDENT
      { "," "normallyConstant" "=" IDENT }
   ")"
   "=" UNSIGNED-NUMBER
```

This annotation has only an effect within a function declaration.

`smoothOrder` defines the 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* $C^{smoothOrder}$.

When a tool computes the derivative of a function, e.g., for index reduction or to compute an ana-

lytic Jacobian, each differentiation of a function reduces the `smoothOrder` *by 1. The* `smoothOrder` *information can then be used to infer continuity of the resulting differentiated function calls, provided the input arguments are continuous. This is a conservative check, however, meaning that a tool may be able to establish continuity of a function call even though the* `smoothOrder` *has been reduced to less than 0, and/or some input arguments are not continuous.*]

The optional argument `normallyConstant` of `smoothOrder` defines that the function argument `IDENT` is usually constant.

[*A tool might check whether the actual argument to* `IDENT` *is a parameter expression at the place where the function is called. If this is the case, the derivative of the function might be constructed under the assumption that the corresponding argument is constant, to enhance efficiency. Typically, a tool would generate at most two different derivative functions of a function: One, under the assumption that all* `normallyConstant` *arguments are actually constant. And one, under the assumption that all input arguments are time varying. Based on the actual arguments of the function call either of the two derivative functions is used.*

This annotation is used by many functions of the `Modelica.Fluid` *library, such as* `Modelica.Fluid.Dissipation.PressureLoss.StraightPipe.dp_laminar_DP`*, since geometric arguments to these functions are usually constant.*]

Annotation 12.2 `derivative`

```
"derivative" [ derivative-constraints ] "=" name

derivative-constraints :
   "(" derivative-constraint { "," derivative-constraint } ")"

derivative-constraint :
   "order" = UNSIGNED-NUMBER
   | "noDerivative" = IDENT
   | "zeroDerivative" = IDENT
```

This annotation has only an effect within a function declaration.

The `derivative` annotation can influence simulation time and accuracy, can be applied to both functions written in Modelica and to external functions, and may be used several times for the same function declaration.

Each use of the `derivative` annotation points to another *derivative-function* that expresses a derivative of the declared function, and the annotation can state that it is only valid under certain restrictions on the input arguments. These restrictions are defined using the optional attributes `order`, `noDerivative`, and `zeroDerivative`. The `order` may be specified at most once for each `derivative` annotation, must be at least 1, and defaults to 1. Specifying `order` is only considered a restriction if `order` > 1.

For details abouts using the `derivative` annotation, see section 12.7.1.

Annotation 12.3 `inverse`

```
"inverse" "(" function-inverse { "," function-inverse } ")"

function-inverse :
   IDENT "=" type-specifier function-call-args"
```

A function with one output formal parameter may have one or more `inverse` annotations to define inverses of this function.

For details abouts using the `inverse` annotation, see section 12.7.3.

12.7.1 Using the Derivative Annotation

The given derivative-function must be a valid derivative if the `derivative` annotation restrictions are satisfied, and can thus be used to compute the derivative in those cases. 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). When a function supplies multiple derivative-functions subject to different restrictions, the first one that can be used (i.e., satisfying the restrictions) will be used for each call.

[*This means that the most restrictive derivatives should be written first.*]

[*Example: The following model illustrates the requirement that a provided derivative must be valid. That* `fder` *is a valid derivative of* `f` *means that it can be used safely to compute* `x2` *by numeric integration: the function value,* `x1`*, will up to numerical precision be matched by the integral of the derivative,* `x2`*.*

```
function f
  input Real x;
  output Real y;
  annotation(derivative = fder);
  external "C";
end f;
model M
  input Real u;
  Real x1 "Directly from function";
  Real x2 "Integrated from derivative";
equation
  x1 = f(u);
  der(x2) = der(x1);
initial equation
  x2 = x1;
end M;
```

Note that tools are not required to use the provided derivative, and might solve the equations completely without numeric integration.]

[*Example: Use of* `order` *to specify a second order derivative:*

```
function foo0 annotation(derivative = foo1);
end foo0;

function foo1 annotation(derivative(order=2) = foo2);
end foo1;

function foo2 end foo2;
```

]

The inputs and outputs of 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. These common inputs must have the same name, type, and declaration order for the function and its derivative.
- The outputs are constructed by starting with an empty list and then in order appending one derivative for each output containing reals. The outputs must have the same type and declaration order for the function and its derivative.

If the Modelica function call is a nth derivative ($n \geq 1$), i.e., this function call has been derived from an $(n-1)$th derivative by differentiation inside the tool, an `annotation(derivative(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 nth `order` derivatives.
- The output arguments are similar to the output argument for the nth derivative, but each output is one higher in derivative order. The outputs must have the same type and declaration order for the function and its derivative.

[*The restriction that only the result of differentiation can use higher-order derivatives ensures that the derivatives* `x`*,* `der_x`*, ... are in fact derivatives of each other. Without that restriction we would have both* `der(x)` *and* `x_der` *as inputs (or perform advanced tests to verify that they are the same).*]

[*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

$$(\ldots, y(t), \ldots) = \mathsf{foo0}(\ldots, x(t), b, \ldots)$$

implies that:

$$(\ldots, \frac{\mathrm{d}y(t)}{\mathrm{d}t}, \ldots) = \mathsf{foo1}(\ldots, x(t), b, \ldots, \ldots, \frac{\mathrm{d}x(t)}{\mathrm{d}t}, \ldots)$$
$$(\ldots, \frac{\mathrm{d}^2y(t)}{\mathrm{d}t^2}, \ldots) = \mathsf{foo2}(\ldots, x(t), b, \ldots, \frac{\mathrm{d}x(t)}{\mathrm{d}t}, \ldots, \ldots, \frac{\mathrm{d}^2x(t)}{\mathrm{d}t^2}, \ldots)$$

]

An input or output to the function may be any simple type (`Real`, `Boolean`, `Integer`, `String` and enumeration types) or a record. For a record containing `Real` values, the corresponding derivative uses a derivative record that only contains the real-predefined types and sub-records containing reals (handled recursively) from the original record. When using `smoothOrder`, then the derivative record is automatically constructed. The function must have at least one input containing reals. The output list of the derivative function shall not be empty.

[*Example: Here is one example use case with records mixing* `Real` *and non-*`Real` *as inputs and outputs:*

```
record ThermodynamicState "Thermodynamic state"
  SpecificEnthalpy h "Specific enthalpy";
  AbsolutePressure p "Pressure";
  Integer phase(min = 1, max = 2, start = 1);
end ThermodynamicState;
```

```
record ThermoDynamicState_der "Derivative"
  SpecificEnthalpyDerivative h "Specific enthalphy derivative";
  PressureDerivative p "Pressure derivative";
  // Integer input is skipped
end ThermodynamicState_der;

function density
  input ThermodynamicState state "Thermodynamic state";
  output Density d "Density";
algorithm
  ...
  annotation(derivative = density_der);
end density;

function density_der
  input ThermodynamicState state "Thermodynamic state";
  input ThermodynamicState_der state_der;
  output DensityDerivative d "Density derivative";
algorithm
  ...
end density_der;

function setState_ph
  input Pressure p;
  input SpecificEnthalpy h;
  input Integer phase = 0;
  output ThermodynamicState state;
algorithm
  ...
  annotation(derivative = setState_ph_der);
end setState_ph;

function setState_ph_der
  input Pressure p;
  input SpecificEnthalpy h;
  input Integer phase;
  input PressureDerivative p_der;
  input SpecificEnthalpyDerivative h_der;
  output ThermodynamicState_der state_der;
algorithm
  ...
end setState_ph_der;

ThermodynamicState state1 = setState_ph(p=..., h=..., phase=...);
Density rho1 = density(state1);
DensityDerivative d_rho1 = der(rho1);
Density rho2 = density(setState_ph(p=..., h=..., phase=...));
DensityDerivative d_rho2 = der(rho2);
```

]

- `"zeroDerivative"` `"="` *inputVar*$_1$ { `","` `"zeroDerivative"` `"="` *inputVar*$_2$ }

The derivative function is only valid if *inputVar*$_1$ (and *inputVar*$_2$ etc.) are independent of the variables the function call is differentiated with respect to (i.e., that the derivative of *inputVar*$_1$ is zero). The derivative of *inputVar*$_1$ (and *inputVar*$_2$ etc.) are excluded from the argument list of the derivative-function. If the derivative-function also specifies a derivative the common variables should have consistent `zeroDerivative`.

[*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 =* `fGeneralDer` *is optional and can be used when the derivative of the matrix or offset is non-zero). Note that the derivative annotation of* `fDer` *must specify* `zeroDerivative` *for both* `y` *and* `offset` *as below, but the derivative annotation of* `fGeneralDer` *shall not have* `zeroDerivative` *for either of them (it may specify*

zeroDerivative *for* x_der, y_der, *or* offset_der*).*

```
function f "Simple table lookup"
  input Real x;
  input Real y[:, 2];
  input Real offset "Shortened to o below";
  output Real z;
algorithm
  ...
  annotation(derivative(zeroDerivative=y, zeroDerivative=offset) = fDer,
             derivative = fGeneralDer);
end f;

function fDer "Derivative of simple table lookup"
  input Real x;
  input Real y[:, 2];
  input Real offset;
  input Real x_der;
  output Real z_der;
algorithm
  ...
  annotation(
    derivative(zeroDerivative=y, zeroDerivative=offset, order=2) = fDer2);
end fDer;

function fDer2 "Second derivative of simple table lookup"
  input Real x;
  input Real y[:, 2];
  input Real offset;
  input Real x_der;
  input Real x_der2;
  output Real z_der2;
algorithm
  ...
end fDer2;

function fGeneralDer "Derivative of table lookup taking
into account varying tables"
  input Real x;
  input Real y[:, 2];
  input Real offset;
  input Real x_der;
  input Real y_der[:, 2];
  input Real offset_der;
  output Real z_der;
algorithm
  ...
  //annotation(derivative(order=2) = fGeneralDer2);
end fGeneralDer;
```

In the example above zeroDerivative=y *and* zeroDerivative=offset *imply that*

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}t}\mathtt{f}(x(t), y(t), o(t)) &= \frac{\partial \mathtt{f}}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}t} + \frac{\partial \mathtt{f}}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}t} + \frac{\partial \mathtt{f}}{\partial o}\frac{\mathrm{d}o}{\mathrm{d}t} \\ &= \frac{\partial \mathtt{f}}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}t} + \frac{\partial \mathtt{f}}{\partial y}\cdot 0 + \frac{\partial \mathtt{f}}{\partial o}\cdot 0 \\ &= \frac{\partial \mathtt{f}}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}t} \\ &= \mathtt{fDer}\cdot\frac{\mathrm{d}x}{\mathrm{d}t}\end{aligned}$$

]

- "noDerivative" "=" $inputVar_1$

The derivative of $inputVar_1$ is excluded from the argument list of the derivative-function. This relies on assumptions on the arguments to the function; and the function should document these assumptions (it is not always straightforward to verify them). In many cases even the undifferentiated function will only behave correctly under these assumptions.

The inputs excluded using `zeroDerivative` or `noDerivative` may be of any type (including types not containing reals).

[*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;
algorithm
  ...
  annotation(derivative(noDerivative=y) = h);
end f;

function h
  input Real x;
  input Real y;
  input Real x_der;
  output Real z_der;
algorithm
  ...
end h;
```

This is useful if `g` *represents the major computational effort of* `fg`.

Therefore `h` *indirectly includes the derivative with respect to* `y` *as follows:*

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}t}\mathtt{fg}(x(t)) &= \frac{\mathrm{d}}{\mathrm{d}t}\mathtt{f}(x(t),\, \mathtt{g}(x(t))) \\ &= \frac{\partial \mathtt{f}}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}t} + \frac{\partial \mathtt{f}}{\partial y}\frac{\partial \mathtt{g}}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}t} \\ &= \left(\frac{\partial \mathtt{f}}{\partial x} + \frac{\partial \mathtt{f}}{\partial y}\frac{\partial \mathtt{g}}{\partial x}\right)\frac{\mathrm{d}x}{\mathrm{d}t} \\ &= \mathtt{h}(x(t),\, y(t)))\frac{\mathrm{d}x}{\mathrm{d}t}\end{aligned}$$

]

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, and the looked up name must be a function), and partially differentiated with respect to each `IDENT` in order (starting from the first one). Each `IDENT` must be a scalar `Real` input 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;
```

]

12.7.3 Using the Inverse Annotation

If a function f_1 with one output formal parameter y can be restricted to an informally defined domain and codomain, such that the mapping of the input formal parameter u_k to y is bijective for any fixed assignment to the other input formal parameters in the domain (see examples below), then it can be given an **inverse** annotation to provide an explicit inverse f_2 to this mapping, provided that the function is only applied on this domain:

The **inverse** annotation takes the following form in a function declaration:

```
function f1
  input A1 u1;
  ...
  input Tk uk;
  ...
  input Am um = am;
  ...
  input An un;
  output T2 y;
algorithm
  ...
  annotation(inverse(uk = f2(..., y, ...));
end f1;
```

In addition to y, the formal call to f_2 in the annotation shall also pass the other formal parameters (excluding u_k) needed determine the inverse, see below. The function f_2 must be an actual inverse, meaning that if u_k is calculated as $u_k = f_2(\ldots, y, \ldots)$, then the equality $y = f_1(\ldots, u_k, \ldots)$ is satisfied up to a certain precision, for all values of the input arguments of $f_2(\ldots, y, \ldots)$ in the range and informal domain of f_1.

More than one inverse can be defined within the same **inverse** annotation, separated by commas:

```
annotation(inverse(uk = f2(..., y, ...), ui = f3(..., y, ...), ...));
```

Function f_1 can have any number and types of formal parameters with and without default value. The restriction is that the *number of unknown variables* (see section 4.8) in the output formal parameter of both f_1 and f_2 must be the same and that f_2 should have a union of output and formal parameters that is the same or a subset of that union for f_1, but the order of the formal parameters may be permuted.

[*Example: Inverse function with same union of formal parameters:*

```
function h_pTX
  input Real p "pressure";
  input Real T "temperature";
```

```
  input Real X[:] "mass fractions";
  output Real h "specific enthalpy";
algorithm
  ...
  annotation(inverse(T = T_phX(p, h, X)));
end h_pTX;

function T_phX
  input Real p "pressure";
  input Real h "specific enthalpy";
  input Real X[:] "mass fractions";
  output Real T "temperature";
algorithm
  ...
end T_phX;
```

]

The subset case is useful if f_1 computes the inverse of f_2 within a region, or up to a certain tolerance. Then, f_1 may specify f_2 as inverse with fewer arguments, skipping the arguments for tolerance and/or the region.

[*Example: Inverse function with subset of formal parameters:*

```
function inv_sine
  input Real x;
  input Real angleOrig;
  output Real angle;
  // Finds sine(angle) = x with angle closest to angleOrig.
algorithm
  ...
  annotation(inverse(x = sine(angle)));
end inv_sine;

function sine
  input Real angle;
  output Real x;
algorithm
  x := sin(angle);
  // Note: No inverse.
end sine;
```

]

Tools are not expected to verify the bijectiveness requirement, meaning that it is the user's responsibility to ensure that this requirement is fulfilled, and that tools can rely on the requirement as an assumption for symbolic manipulations when an inverse function is provided.

There is no guarantee that a provided inverse will be used, and no rule for at which stage of symbolic processing it could be applied. Inlining a function means that the possibility to apply provided inverses is lost. Hence, the recommended inlining annotations – if any – for use together with the `inverse`-annotation is either `Inline = false` or `LateInline = true`.

[*Example: If an inverse is provided, but the injectiveness part of the bijectiveness requirement is not fulfilled, this may introduce additional ambiguity to the solution of equations with multiple solutions. Consider the following invalid use of the* `inverse` *annotation:*

```
model NotInjective
  function square
    input Real x;
    output Real y = x^2;
    annotation(inverse(x = sqrt(y))); // Invalid!}
  end square;

  parameter Real y0 = -1.0;
```

```
  Real y(start = y0, fixed = true);
  Real x(start = sign(y0) * sqrt(abs(y0))); // Good guess with same sign as y.
equation
  der(y) = -y;
  square(x) = abs(y); // Expecting continuous solution for x.
end NotInjective;
```

That the parameter `y0` *may have any sign means the sign of* `x` *cannot be restricted in the informal domain of* `square`*, and hence that the injectiveness requirement cannot be fulfilled. Without the* `inverse` *annotation, the nonlinear equation in* `x` *and* `y` *has an ambiguity, but it is generally expected that this is handled so that a continuous solution for* `x` *is obtained, meaning that it will keep the same sign as* `y` *throughout the simulation. The additional ambiguity introduced by the* `inverse` *annotation is that if the provided inverse is used to solve the nonlinear equation instead of using a generic nonlinear equation solver based on local search, then the solution with positive sign is always obtained. The lack of guarantees that a provided inverse will be used thus implies a worse ambiguity than what was present in the model before introducing the* `inverse` *annotation.*]

[*Example: If an inverse is provided, but the surjectiveness part of the bijectiveness requirement is not fulfilled, this may introduce an invalid solution to equations that do not have a solution at all. Consider the following invalid use of the* `inverse` *annotation:*

```
model NotSurjective
  function cube
    input Real x;
    output Real y = x ^ 3;
  end cube;

  function cbrtPos "Cubic root of positive number"
    input Real y;
    output Real x;
  algorithm
    assert(y > 0, "Argument must be positive.");
    x := exp(log(y) / 3);
    annotation(inverse(y = cube(x))); // Invalid!}
  end cbrtPos;

  Real x = 0.5 + sin(time);
  Real y;
equation
  cbrtPos(y) = x; // Calling cbrtPos requires y > 0.
  annotation(experiment(StopTime = 10.0));
end NotSurjective;
```

As the value of `x` *varies over the interval* $[-1, 1]$*, but the range of* `cbrtPos` *is only* $(0, \infty)$*, the informal codomain of* `cbrtPos` *cannot be restricted such that the surjectiveness is fulfilled. A valid solution to the equation in* `x` *and* `y` *must satisfy* $y > 0$*, and when no* `inverse` *annotation is given, a violation will be detected by a nonlinear solver applied directly to the equation. When the (invalid) inverse provided by the* `inverse` *annotation is used, however, the equation gets transformed into*

```
  y = cube(x);
```

where the requirement $y > 0$ *can no longer be detected, resulting in a simulation result that does not fulfill the original model equations.*]

12.8 Function Inlining and Event Generation

The annotations listed below affect inlining of functions and the related topic of event generation inside functions.

Annotation	*Description*	*Details*
`Inline`	Inline function	Annotation 12.4
`LateInline`	Inline after all symbolic transformations	Annotation 12.5
`InlineAfterIndexReduction`	Inline after index reduction	Annotation 12.6
`GenerateEvents`	Generate events for zero crossings in function	Annotation 12.7

Inlining a function makes the statements of the function body accessible to symbolic operations, potentially leading to expression simplifications and more efficient solution of equations. At the same time, another important consequence of inlining a function is that any annotations for derivatives or inverses are lost. Hence, one needs to find the right balance between inlining too early (loss of provided derivatives and inverses) and too late (earlier stages of symbolic processing cannot benefit from symbolic simplifications).

Annotation 12.4 `Inline`

```
"Inline" "=" ( false | true )
```

Has only an effect within a function declaration.

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.

If `Inline = false`, the model developer proposes to not inline the function.

[`Inline = true` *is for example used in* `Modelica.Mechanics.MultiBody.Frames` *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.*]

Annotation 12.5 `LateInline`

```
"LateInline" "=" ( false | true )
```

Has only an effect within a function declaration.

If `LateInline = true`, the model developer proposes to inline the function after all symbolic transformations have been performed.

[*Late inlining is especially useful for differentiation and inversion of functions; for efficiency reasons it is then useful to replace all function calls with identical input arguments by one function call, before the inlining.*]

If `LateInline = false`, the model developer proposes to not inline the function after symbolic transformations have been performed.

`Inline = true, LateInline = false` is identical to `Inline = true`.

`Inline = true, LateInline = true` is identical to `LateInline = true`.

`Inline = false, LateInline = true` is identical to `LateInline = true`.

[*This annotation is for example used in* `Modelica.Media.Water.IF97_Utilities.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.*]

Annotation 12.6 `InlineAfterIndexReduction`

```
"InlineAfterIndexReduction" "=" ( false | true )
```

Has only an effect within a function declaration.

If `true`, the model developer proposes to inline the function after the function is differentiated for index reduction, and before any other symbolic transformations are performed. This annotation cannot be combined with annotations `Inline` and `LateInline`.

Annotation 12.7 `GenerateEvents`

```
"GenerateEvents" "=" ( false | true )
```

Has only an effect within a function declaration.

By default, `GenerateEvents = false` and expressions in the function body that would normally be event-generating shall not generate events, similar to inlining the function body while wrapping all expressions in `noEvent`, see operator 3.21. By specifying `GenerateEvents = true`, event-generating expressions in the function body shall generate events as normal, similar to inlining the function body without wrapping all expressions in `noEvent`. Having `GenerateEvents = true` implies `Inline = true` unless overridden by specifying one of the inlining annotations with value `true` (in particular, `GenerateEvents = true` cannot be combined with `Inline = false`).

[*In case a given inlining annotation proposes to inline at a stage when the tool cannot process* `GenerateEvents = true`*, it is recommended to give a diagnostic and instead perform inlining of the function at the nearest stage where* `GenerateEvents = true` *can still be respected.*

If the function is called in a context where events will not be generated (e.g., inside another function without `GenerateEvents = true`*) no special action is needed.*]

12.9 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 (specifically C89) and FORTRAN 77. Other languages, e.g., C++ and Fortran 90, may be supported in the future, and provided the function is link-compatible with C89 or FORTRAN 77 it can be written in any language.
- 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, by passing an output that has a binding equation to the external function.

  [*Binding equations are executed prior to calling the external function.*]

The format of an external function declaration is as follows.

```
function IDENT description-string
  { component-clause ";" }
  [ protected { component-clause ";" } ]
external [ language-specification ]
  [ external-function-call ]
    [ annotation-clause ] ";"
  [ annotation-clause ";" ]
end IDENT;
```

Just as for any other function, components in the public part of an external function declaration shall be declared either as `input` or `output`.

Protected components can be passed to the external function without being initialized by means of a declaration equation, which is useful for passing workspace memory to functions with FORTRAN style memory management, and the reason for passing them in the same (writable) way as output components (see section 12.9.1). The value of a protected component passed to the external function should be considered undefined (destroyed) after the external function call.

The *`language-specification`* must currently be one of `"builtin"` (deprecated), `"C"`, `"C..."` (for one of the specific C standards like C89, C99, and C11 – specifying that it relies on the C standard library of that version) or `"FORTRAN 77"`. Unless the external language is specified, it is assumed to be `"C"`.

[*The intended use of, e.g., C99 is to detect if the user tries to link with a C99-function using a C89 compiler.*]

The deprecated **"builtin"** specification is only used for the elementary mathematical functions described in section 3.7.3. The external function call mechanism for **"builtin"** functions is implementation-defined.

[*Typically, for functions from the standard C library, the prototype of the function is provided but no* `Library` *annotation. Currently, there are no other builtin functions defined in Modelica.*]

[*Example:*

```
package Modelica
  package Math
    function sin
      input Real x;
      output Real y;
    external "builtin"
      y = sin(x);
    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 component references, scalar constant expressions, 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.

A component reference to a component that is part of an input or output is treated the same way as a top-level input or output in the external call.

12.9.1 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` calls are passed as inputs.

12.9.1.1 Simple Types

Arguments of *simple* types are by default mapped as follows for C:

Modelica	*C*	
	Input	*Output*
`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'`) and are encoded using UTF-8 (assuming `CHAR_BIT ==8` in C) to facilitate calling of C functions. The valid return values for an external function returning a `String` are:

- A string given as `String` input to the external function.

- A pointer to a C string literal.
- A pointer returned by one of the string allocation functions in section 12.9.6.2.

[*The reason why it is not allowed to return a string allocated with, for instance,* `malloc` *is that there is no transfer of ownership when a string is returned from the external function. The external code would remain the owner of such a string, and would be responsible for eventually releasing the memory at some point. Consequently, the Modelica simulation environment would not be able to assume that only its own string deallocation routines could invalidate any of the strings returned by external functions.*]

`Boolean` values are mapped to C such that `false` in Modelica is 0 in C and `true` in Modelica is 1 in C. If the returned value from C is 0 it is treated as `false` in Modelica; otherwise as `true`.

[*It is recommended that the C function should interpret any non-zero value as true.*]

Arguments of simple types are by default mapped as follows for FORTRAN 77:

Modelica	*FORTRAN 77*	
	Input	*Output*
`Real`	`DOUBLE PRECISION`	`DOUBLE PRECISION`
`Integer`	`INTEGER`	`INTEGER`
`Boolean`	`LOGICAL`	`LOGICAL`
`String`	*Special*	*Not available*
Enumeration type	`INTEGER`	`INTEGER`

Sending string literals to FORTRAN 77 subroutines/functions is supported for LAPACK/BLAS-routines, and the strings are NUL-terminated for compatibility with C. String are UTF-8 encoded, even though the support for non-ASCII characters in FORTRAN 77 is unclear and it is not relevant for the LAPACK/BLAS-routines. Returning strings from 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 ith enumeration literal is mapped to integer value i, starting at 1.

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.

12.9.1.2 Arrays

Unless an explicit function call is present in the **`external`**-clause, 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 stored in row-major order when calling C functions and in column-major order when calling FORTRAN 77 functions.

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 12.9.1.1 or a record type as defined in section 12.9.1.3 and it is mapped to the type T' as defined in these sections for input arguments. Array inputs to C-functions are const-pointers, indicating that the arrays shall not be changed.

Modelica	*C*	
	Input	*Output*
`T[`dim_1`]`	`const` T' `*, size_t` dim_1	T' `*, size_t` dim_1
`T[`dim_1`, `dim_2`]`	`const` T' `*, size_t` dim_1`, size_t` dim_2	T' `*, size_t` dim_1`, size_t` dim_2
`T[..., `dim_n`]`	`const` T' `*, ..., size_t` dim_n	T' `*, ..., size_t` dim_n

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 12.9.1.1 and it is mapped to the type T' as defined in that section.

Modelica	*FORTRAN 77* *Input and output*
T[dim_1]	T', INTEGER dim_1
T[dim_1, dim_2]	T', INTEGER dim_1, INTEGER dim_2
T[dim_1, ..., dim_n]	T', INTEGER dim_1, ..., INTEGER dim_n

[*Example: 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(const 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.

[*Example: This example shows how 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
```

]

12.9.1.3 Records

Mapping of record types is only supported for C. A Modelica record class is mapped as follows:

- The record class is represented by a struct in C.
- Each component of the Modelica record is mapped to its corresponding C representation. A nested record component is mapped to a nested struct component.
- The components of the Modelica record class are declared in the same order in the C struct.
- Arrays cannot be mapped.

Records are passed by reference (i.e., a pointer to the record is being passed).

[*Example:*

```
record A
  Integer b;
end A;
record R
  Real x;
  Real z;
  A a1, a2;
end R;
```

is mapped to:

```
struct A {
  int b;
};
struct R {
  double x;
  double z;
  struct A a1, b2;
};
```

]

12.9.2 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. Otherwise the external function is assumed not to return anything; i.e., it is really a procedure or, in C, a void-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. See section 12.9.1.1 regarding returning of `String` values.

[*In the case of an external function not returning anything, argument type mapping according to section 12.9.1.1 is performed in the absence of any explicit external function call.*]

Return types are by default mapped as follows for C and FORTRAN 77:

Modelica	*C*	*FORTRAN 77*
`Real`	`double`	`DOUBLE PRECISION`
`Integer`	`int`	`INTEGER`
`Boolean`	`int`	`LOGICAL`
`String`	`const char*`	*Not allowed*
`T`[dim_1, ..., dim_n]	*Not allowed*	*Not allowed*
Enumeration type	`int`	`INTEGER`
Record	See section 12.9.1.3	*Not allowed*

The element type `T` of an array can be any simple type as defined in section 12.9.1.1 or, for C, a record type is returned as a value of the record type defined in section 12.9.1.3.

12.9.3 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.*]

12.9.4 Annotations for External Functions

The following annotations are useful in the context of calling external functions from Modelica, and they should occur on the **`external`**-clause and no other standard annotations should occur on the **`external`**-clause. They can all specify either a scalar value or an array of values as indicated below for the `Library` annotation:

- 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. For shared libraries it is recommended to include all non-system libraries in this list.
- The **`annotation`**`(Include="insertedCode")`, used to insert function prototypes or definitions needed for calling the external function in the code generated by the Modelica compiler. When generating a call to the external function, the `"insertedCode"` shall be present at the top level somewhere before the point of the call (similar to where include directives are typically placed). The `Include` annotation shall be used in such a way that each external function can be handled in a separate translation unit. In particular, different external functions must not have `Include` annotations providing exported definitions of the same function symbol to avoid linking errors.

  A deprecated feature is that if multiple `Include` annotations – possibly coming from different external functions – have identical content, the tool shall not incude this content more than once in any translation unit. In case calls to several external functions are generated in the same translation unit, the `Include` annotations of the different functions must not define the same function – except when relying on the deprecated behavior.

  The included code should be valid C89 code.

  When an `Include` annotation is present, it shall provide a prototype for the external function, and hence the tool shall not produce an automatically generated prototype in the generated code in this case.

  Although all pointer types are const pointers in the type mapping for input arguments, it is a deprecated feature that the prototype in an `Include` annotation may use non-const pointers instead.

  [*For an external function declaration calling the external function* `myfoo`*, examples of* `"insertedCode"` *include:*

  - *An* `#include` *directive including a header file with a prototype for* `myfoo`.
  - *An* `#include` *directive including a source file with a* `static` *definition of* `myfoo`. *Include guards should be used (either in the* `Include` *annotation or in the source file) to avoid relying on the deprecated feature that tools shall include at most one copy in the same translation unit. (Having a* `static` *definition allows the same source file to be included by multiple* `Include` *annotations in different translation units.)*
  - *A prototype for* `myfoo`. *This may be useful when no header file is available and it is not desirable to rely on the automatic generation of a prototype.*

  - *A piece of C code directly defining* `myfoo`. *Since no other* `Include` *annotation is expected to contain a definition of* `myfoo`*, it is not necessary to make the definition* `static`.

]

- The `annotation(IncludeDirectory="modelica:/ModelicaLibraryName/Resources/Include")`, used to specify a location for header files. The preceding one is the default and need not be specified; but another location could be specified by using an URI name for the include directory, see section 13.5.
- The `annotation(LibraryDirectory="modelica:/ModelicaLibraryName/Resources/Library")`, used to specify a location for library files. The preceding one is the default and need not be specified; but another location could be specified by using an URI name for the library directory, see section 13.5. Different versions of one object library can be provided (e.g., for Windows and for Linux) by providing a *platform* directory below the `LibraryDirectory`. If no platform directory is present, the object library must be present in the `LibraryDirectory`. The following *platform* names are standardized:
  - `"win32"` (Microsoft Windows 32 bit)
  - `"win64"` (Microsoft Windows 64 bit)
  - `"linux32"` (Linux Intel 32 bit)
  - `"linux64"` (Linux Intel 64 bit)
- The `annotation(SourceDirectory="modelica:/ModelicaLibraryName/Resources/Source")`, gives the location for source files. The preceding one is the default and need not be specified; but another location could be specified by using an URI name for the source directory, see section 13.5. It is not specified how they are built.

The win32 or win64 directories may contain gcc47, vs2010, vs2012 for specific versions of these compilers and these are used instead of the general win32 or win64 directories, and similarly for other platforms.

The library on Windows may refer to a lib-file (static library), both a lib- and dll-file (in this case the lib-file is an import-library), or just a dll-file. It shall not refer to an obj-file.

If the directory for the specific compiler version is missing the platform specific directory is used.

[*A tool may give a diagnostic if the directory corresponding to the selected compiler version is missing. The directories may use symbolic links or use a text-file as described below: e.g., a text-file vs2008 containing the text ../*win32/vs2005 *(or* vs2005*) suggesting that it is compatible with vs2005.*]

The `ModelicaLibraryName` used for `IncludeDirectory`, `LibraryDirectory`, and `SourceDirectory` indicates the top-level class where the annotation is found in the Modelica source code.

[*Example: Use of external functions and of object libraries:*

```
package ExternalFunctions
  model Example
    Real x(start = 1.0), y(start = 2.0);
  equation
    der(x) = -ExternalFunc1(x);
    der(y) = -ExternalFunc2(y);
  end Example;

  model OtherExample
    Real x(start = 1.0);
  equation
    der(x) = -ExternalFunc3(x);
  end OtherExample;

  function ExternalFunc1 "Include header file for library implementation"
    input Real x;
    output Real y;
  external "C"
    y = ExternalFunc1_ext(x)
```

```
      annotation(Library = "ExternalLib1",
                 Include = "#include \"ExternalFunc1.h\"",
                 SourceDirectory =
                  "modelica:/ExternalFunctions/Resources/Source");
      // The specified SourceDirectory is the default and thus redundant.
  end ExternalFunc1;

  function ExternalFunc2 "Include header file for library implementation"
    input Real x;
    output Real y;
  external "C"
      annotation(Library = "ExternalLib2",
                 Include = "#include \"ExternalFunc2.h\"");
  end ExternalFunc2;

  function ExternalFunc3 "Include source file"
    input Real x;
    output Real y;
  external "C"
      annotation(Include = "#include \"ExternalFunc3.c\"");
  end ExternalFunc3;
end ExternalFunctions;

package MyExternalFunctions
  extends ExternalFunctions;
end MyExternalFunctions;
```

Directory structure:

ExternalFunctions	
package.mo	– Modelica code from above
Resources	
Include	– Include files
ExternalFunc1.h	– C header file
ExternalFunc2.h	– C header file
ExternalFunc3.c	– C source file (not ideal)
Library	– Object libraries for different platforms
win32	
ExternalLib1.lib	– Static link library for VisualStudio
ExternalLib2.lib	statically linking the dynamic link library
ExternalLib2.dll	– Dynamic link library (with manifest)
linux32	
libExternalLib1.a	– Static link library
libExternalLib2.so	– Shared library
Source	– Sources for library
Func1.c	– C source for ExternalLib1.lib
Func2.c	– C source for ExternalLib2.lib
HelperFunc.c	– C source also included in ExternalLib2.lib
MyExternalFunctions	
package.mo	

Note that calling the function `MyExternalFunctions.ExternalFunc1` *will use the header and library files from* `ExternalFunction`*, the* `ExternalFunctions.Example` *will not use ExternalFunc3.c, and one library file may contain multiple functions.*

The C-source ExternalFunc3.c will be included fully, and is not part of any library. That is not ideal for C-code, but it works for small functions.

It is not specified how the C-sources in the specified `SourceDirectory` *will be used to build the libraries.*

Header file for the function in the dynamic link / shared library ExternalLib2 so that the desired functions are defined to be exported for Microsoft VisualStudio and for GNU C compiler (note, for Linux it is recommended to use the compiler option `-fPIC` *to build shared libraries or object libraries that are later*

transformed to a shared library):

```
/* File ExternalFunc2.h */
#ifndef EXTERNAL_FUNC2_H_
#define EXTERNAL_FUNC2_H_
#ifdef __cplusplus
extern "C" {
#endif
#ifdef _MSC_VER
#ifdef EXTERNAL_FUNCTION_EXPORT
#  define EXTLIB2_EXPORT __declspec( dllexport )
#else
#  define EXTLIB2_EXPORT __declspec( dllimport )
#endif
#elif  __GNUC__ >= 4
  /* In gnuc, all symbols are by default exported. It is still often useful,
  to not export all symbols but only the needed ones */
#  define EXTLIB2_EXPORT __attribute__ ((visibility("default")))
#else
#  define EXTLIB2_EXPORT
#endif

EXTLIB2_EXPORT double ExternalFunc2(double);

#ifdef __cplusplus
}
#endif
#endif
```

]

The `Library` name and the `LibraryDirectory` name in the function annotation are mapped to a linkage directive in a compiler-dependent way thereby selecting the object library suited for the respective computer platform.

12.9.5 Examples

12.9.5.1 Input Parameters, Function Value

[*Example: 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);
```

]

12.9.5.2 Arbitrary Placement of Output Parameters, No External Function Value

[*Example: 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);
```

]

12.9.5.3 Both Function Value and Output Variable

[*Example: 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);
```

]

12.9.6 Utility Functions

This section describes the utility functions declared in ModelicaUtilities.h, which can be called in external Modelica functions written in C.

12.9.6.1 Error Reporting Utility Functions

The functions listed below produce a message in different ways.

Expression	*Description*	*Details*
`ModelicaMessage(`*string*`)`	Message with fixed string	
`ModelicaWarning(`*string*`)`	Warning with fixed string	Function 12.1
`ModelicaError(`*string*`)`	Error with fixed string	
`ModelicaFormatMessage(`*format*`, ...)`	Message with `printf` style formatting	
`ModelicaFormatWarning(`*format*`, ...)`	Warning with `printf` style formatting	Function 12.2
`ModelicaFormatError(`*format*`, ...)`	Error with `printf` style formatting	
`ModelicaVFormatMessage(`*format*`, `*ap*`)`	Message with `vprintf` style formatting	
`ModelicaVFormatWarning(`*format*`, `*ap*`)`	Warning with `vprintf` style formatting	Function 12.3
`ModelicaVFormatError(`*format*`, `*ap*`)`	Error with `vprintf` style formatting	

The *Message*-functions only produce the message, but the *Warning*- and *Error*-functions combine this with error handling as follows.

The *Warning*-functions view the message as a warning and can skip duplicated messages similarly as an `assert` with `level = AssertionLevel.Warning` in the Modelica code.

The *Error*-functions never return to the calling function, but handle the error similarly to an `assert` with `level = AssertionLevel.Error` in the Modelica code.

Function 12.1 `ModelicaMessage`, `ModelicaWarning`, `ModelicaError`

```
void ModelicaMessage(const char* string)
void ModelicaWarning(const char* string)
void ModelicaError(const char* string)
```

Output the fixed message string (no format control).

Function 12.2 `ModelicaFormatMessage`, `ModelicaFormatWarning`, `ModelicaFormatError`

```
void ModelicaFormatMessage(const char* format, ...)
void ModelicaFormatWarning(const char* format, ...)
void ModelicaFormatError(const char* format, ...)
```

Output the message under the same format control as the C function `printf`.

Function 12.3 `ModelicaVFormatMessage`, `ModelicaVFormatWarning`, `ModelicaVFormatError`

```
void ModelicaVFormatMessage(const char* format, va_list ap)
void ModelicaVFormatWarning(const char* format, va_list ap)
void ModelicaVFormatError(const char* format, va_list ap)
```

Output the message under the same format control as the C function `vprintf`.

12.9.6.2 String Allocation Utility Functions

The functions listed below are related to string allocation.

Expression	*Description*	*Details*
`ModelicaAllocateString(`*len*`)`	Allocate or error	Function 12.4
`ModelicaAllocateStringWithErrorReturn(`*len*`)`	Allocate or null	Function 12.5
`ModelicaDuplicateString(`*str*`)`	Duplicate or error	Function 12.6
`ModelicaDuplicateStringWithErrorReturn(`*str*`)`	Duplicate or null	Function 12.7

As described in section 12.9.1.1, an external function wanting to return a newly constructed string must allocate this string with one of the string allocation functions in this section. The allocated memory is owned by the Modelica simulation environment, and may only be accessed by the external function during the currently executing external function call. The string allocation functions can also be used to allocate temporary strings that are not returned from the external function, with the convenience of the Modelica simulation environment being responsible for deallocation after the return of the external function. (This is particularly convenient for avoiding memory leaks in the event of abnormal termination of the external function, for example, via `ModelicaError`).

[*Memory that is not passed to the Modelica simulation environment, such as memory that is freed before leaving the function, or in an* `ExternalObject`*, see section 12.9.7, may be allocated with the standard C mechanisms, like* `malloc`.]

Function 12.4 ModelicaAllocateString

```
char* ModelicaAllocateString(size_t len)
```

Allocates $len + 1$ characters, and sets the last one to NUL. If an error occurs, this function does not return, but calls `ModelicaError`.

Function 12.5 ModelicaAllocateStringWithErrorReturn

```
char* ModelicaAllocateStringWithErrorReturn(size_t len)
```

Same as `ModelicaAllocateString`, 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 `ModelicaError` or `ModelicaFormatError` to signal the error.

Function 12.6 ModelicaDuplicateString

```
char* ModelicaDuplicateString(const char* str)
```

Returns a writeable duplicate of the NUL-terminated string *str*. If an error occurs, this function does not return, but calls `ModelicaError`.

Function 12.7 ModelicaDuplicateStringWithErrorReturn

```
char* ModelicaDuplicateStringWithErrorReturn(const char* str)
```

Same as `ModelicaDuplicateString`, 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 `ModelicaError` or `ModelicaFormatError` to signal the error.

12.9.7 External Objects

External functions may need to store their internal memory 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`, may extend from empty classes (definition 4.1), but not contain any other elements. The functions `constructor` and `destructor` shall not be replaceable. It is not legal to call the `constructor` and `destructor` functions explicitly.

- The `constructor` function is called exactly once before the first use of the object. The `constructor` shall have exactly one output argument in which the constructed instance derived from `ExternalObject` is returned. The arguments to the constructor must not – directly nor indirectly – depend on the external object being constructed. The constructor shall initialize the object, and must not require any other calls to be made for the initialization to be complete (e.g., from an initial algorithm or initial equation).

  The constructor shall not assume that pointers sent to the external object will remain valid for the life-time of the external object. An exception is that if the pointer to another external object is given as argument to the constructor, that pointer will remain valid as long as the other external object lives.

- For each completely constructed object, the `destructor` is called exactly once, after the last use of the object, even if an error occurs. The `destructor` shall have no output arguments and the only input argument of the destructor shall be of the type derived from `ExternalObject`. The destructor shall delete the object, and must not require any other calls to be made for the deletion to be complete (e.g., from a `when terminal()` clause).

  [*External objects may be a protected component (or part of one) in a function. The constructor is in that case called at the start of the function call, and the destructor when the function returns, or when recovering from errors in the function.*]

[*External objects may be an input (or part of an input) to a function, in that case the destructor is not called (since the external object is active before and after the function call). Normally this is an external function, but it could be a non-external function as well (e.g., calling external functions one or more times). The function input shall not have a default value using the constructor.*]

- An external object class shall be of the specialized class **class**.

  [*Apart from empty classes (definition 4.1), this is the only use of* **class**.]

- Classes derived from `ExternalObject` can neither be used in an **extends**-clause nor in a short class definition.

- Only the constructor may return external objects and an external object can only be bound in component declarations and neither modified later nor assigned to.

  [*It follows that a function cannot return a component containing an external object, since only the constructor may return an external object and the constructor exactly returns the external object.*]

- 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(const char* fileName, const 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

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.

13.1 Package as Specialized Class

The package concept is a specialized class (section 4.7), using the keyword **package**.

13.2 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.

[*The* **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 syntactic forms:

import *definitionname*; (qualified import of top-level definition)

import *packagename*.*definitionname*; (qualified import)

import *packagename*.{def_1, def_2, ..., def_n}; (multiple definition import)

import *packagename*.*; (unqualified import)

import *shortname* = *definitionname*; (renaming import of top-level definition)

import *shortname* = *packagename*.*definitionname*; (renaming 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. The multiple definition import is equivalent to multiple single definition imports with corresponding *packagename* and definition names.

13.2.1 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 deviates from the normal lexical lookup. For example, consider `A.B.C` in the **import**-clauses **import** `A.B.C;`, **import** `D = A.B.C;`, or **import** `A.B.C.*;`. Here, lookup starts with the lexical lookup of the first part of the name (`A`) 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.

[*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 shortname 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 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.{Add,Sub}; // Accessed by Add and Sub
import Modelica.Math.ComplexNumbers.*;      // Accessed by Add
import Co = Modelica.Math.ComplexNumbers;   // Accessed by Co.Add
```

]

13.2.2 Rules for Import-Clauses

The following rules apply to **import**-clauses:

- The **import**-clauses are *not* inherited.
- The **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 shall not have the same import name (see section 5.3.1).

13.3 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 order that a user can work in parallel with different Modelica tools, it is advisable to not have this list as environment variable, but as a setting in the respective tool. Since* `MODELICAPATH` *is tool dependent, it is not specified in which way the list of library roots is stored. Typically, on a Windows system* `MODELICAPATH` *is a string with path names separated by ';' whereas on a Linux system it is a string with path names separated by a ':'.*]

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 18.9) 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`.]

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.

[*Example: Figure 13.1 below 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.*

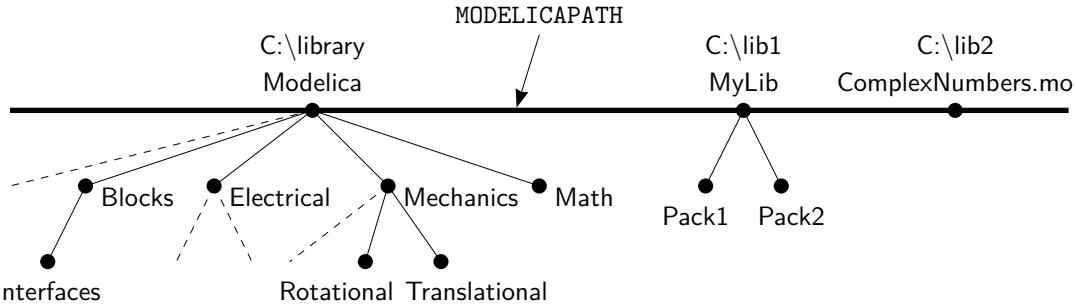


Figure 13.1: *Roots of package hierarchies, e.g.,* `Modelica`, `MyLib`, *and* `ComplexNumbers` *in* `MODELICAPATH` = *"C:\library;C:\lib1;C:\lib2".*

Assume that we want to access the package `MyLib.Pack2` *in figure 13.1 above, 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 name in the import-statement. 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.*]

13.4 File System Mapping of Package/Class

A package/class hierarchy may be represented in the hierarchical structure of the operating system (the file system). For classes with version information see also section 18.9.3. The nature of such an external entity falls into one of the following two groups:

- Directory in the file system.
- File in the file system.

Each Modelica file in the file-system is stored in UTF-8 format (defined by The Unicode Consortium; `https://unicode.org`). A deprecated feature is that the file may start with the UTF-8 encoded BOM (byte order mark; `0xef 0xbb 0xbf`); this is treated as white-space in the grammar. Since the use of BOM is deprecated, tools can ignore any BOM when reading, and it is recommended to never write it.

[*Tools may also store classes in data-base systems, but that is not standardized.*]

13.4.1 Directory Hierarchy Mapping

A directory shall contain a node, 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 directory may also contain one or more sub-entities (directories or files). The sub-entities are mapped as elements of the class defined by their enclosing structured entity. Two sub-entities shall not define classes with identical names

[*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`.]

[*Example: A directory shall not contain both the sub-directory* A *and the file* A.mo.]

In order to preserve the order of sub-entities it is advisable to create a file package.order where each line contains the name of one class or constant (using its Modelica `IDENT` form). If a package.order is present when reading a structured entity the classes and constants are added in this order; if the contents does not exactly match the classes and constants in the package, the resulting order is tool specific and a warning may be given. Classes and constants that are stored in package.mo are also present in package.order but their relative order should be identical to the one in package.mo (this ensures that the relative order between classes and constants stored in different ways is preserved).

13.4.2 Single File Mapping

When mapping a package or class-hierarchy to a file (e.g., the file A.mo), that file shall only define a single class `A` with a name matching the name of the nonstructured entity. In a file hierarchy the files shall have the extension .mo.

A .mo file defining more than one class cannot be part of the mapping to file-structure and it is an error if it is loaded from the `MODELICAPATH`.

13.4.3 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; and this class must exist and must not have been defined using a short class definition.

[*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
  ...
```

]

13.5 External Resources

In order to reference external resources from documentation (such as links and images in html-text) and/or to reference images in the `Bitmap` annotation (see section 18.7.5.6). Absolute URIs should be used, for example file:/// and the URI scheme modelica:/ which can be used to retrieve resources associated with a package. According to the URI specification scheme names are case-insensitive, but the lower-case form should be used, that is Modelica:/ is allowed but modelica:/ is the recommended form.

The Modelica-scheme has the ability to reference a hierarchical structure of resources associated with packages. The same structure is used for all kind of resource references, independent of use (external file, image in documentation, bitmap in icon layer, and link to external file in the documentation), and regardless of the storage mechanism.

Any Modelica-scheme URI containing a slash after the package-name is interpreted as a reference to a resource. The first *segment* of the *path* of the URI is interpreted as a fully qualified package name and the rest of the *path* of the URI is interpreted as the path (relative to the package) of the resource. Each storage scheme can define its own interpretation of the path (but care should be taken when converting from one storage scheme or when restructuring packages that resource references resolve to the same resource). Any storage scheme should be constrained such that a resource with a given path should be unique for any package name that precedes it. The second segment of the path shall not be the name of a class in the package given by the first segment.

As a deprecated feature the URI may start with modelica:// and use the host-part of the authority as the fully qualified package name. That feature is widely used, but deprecated since host-names are generally case-insensitive.

[*Examples of deprecated URIs would be modelica://Modelica/Resources/C.jpg (referring to a resource) and modelica://Modelica.Blocks (referring to a package). These should be rewritten as modelica:/Modelica/Resources/C.jpg and modelica:/Modelica.Blocks.*]

When Modelica packages are stored hierarchically in a file-system (i.e., package `A` in a directory A containing package.mo) the resource modelica:/A/Resources/C.jpg should be stored in the file A/Resources/C.jpg, it is not recommend to use modelica:/A.B/C.jpg for referencing resources; it could be stored in the file A/B/C.jpg – which is counter-intuitive if `A.B` is stored together with `A`. When Modelica packages are stored in other formats a similar mapping should be defined, such that a resource with a given path should be unique for any package name that precedes it. The second segment of the path shall not be the name of a class in the package given by the first segment. As above for Modelica 3.2.1/package.mo, i.e., resources starting from Modelica 3.2.1, and modelica:/Modelica.Mechanics/C.jpg is Modelica 3.2.1/Mechanics/C.jpg – regardless of whether `Modelica.Mechanics` is stored in Modelica 3.2.1/package.mo, Modelica 3.2.1/Mechanics.mo, or Modelica 3.2.1/Mechanics/package.mo.

When mapping a Modelica URI to a file-system path, the file-system path shall end in a directory separator if and only if the URI path ends in the segment separator '/'. For example, if modelica:/A/Resources maps to A/Resources, then modelica:/A/Resources/ maps to A/Resources/, and vice versa.

[*The use of a trailing segment separator is recommended when the resource is a directory and the file-system path will be prepended to relative file paths within the directory. If possible, use URIs for specific files or specific sub-directories instead of appending relative paths to a generic URI such as modelica:/A/Resources/ as the latter creates a dependency on the entire directory.*]

For a Modelica-package stored as a single file, A.mo, the resource modelica:/A/C.jpg refers to a file C.jpg stored in the same directory as A.mo, but using resources in this variant is not recommended since multiple packages will share resources.

In case the name of the class contains quoted identifiers, the single-quote ‘'’ and any reserved characters (‘:’, ‘/’, ‘?’, ‘\#’, ‘[’, ‘]’, ‘@’, ‘!’, ‘\$’, ‘\&’, ‘(’, ‘)’, ‘*’, ‘+’, ‘,’, ‘;’, ‘=’) should be percent-encoded as normal in URIs.

[*Example: Consider a top-level package* `Modelica` *and a class* `Mechanics` *inside it, a reference such as modelica:/Modelica.Mechanics/C.jpg is legal, while modelica:/Modelica/Mechanics/C.jpg is illegal. The references modelica:/Modelica.Mechanics/C.jpg and modelica:/Modelica/C.jpg must also refer to two distinct resources.*]

13.6 Multilingual Descriptions

[*Descriptive texts in a model or library are usually formulated in English. This section describes how a tool can present the library in another language. Translated Modelica text is provided by external files, so that no modification of the original Modelica text is required.*]

The texts in following Modelica constructs should be translated:

- description strings of component declarations and classes
- strings in the following annotations:
  - `Text.string`, `Text.textString`
  - `missingInnerMessage`, `obsolete`, `unassignedMessage`
  - `Dialog.group`, `Dialog.tab`
  - `Dialog.loadSelector.caption`, `Dialog.loadSelector.filter`, `Dialog.saveSelector.caption`, `Dialog.saveSelector.filter`
  - `Documentation.info`, `Documentation.revisions`
  - `Figure.title`, `Figure.caption`, `Figure.group`, `Plot.title`, `Axis.label`, `Curve.legend`
  - `mustBeConnected`

[*None of the translatable constructs can have any impact on simulation results.*]

Comments (delimited as well as rest-of-line) are not translated. Only constructs given entirely by one or more concatenated string literals are translated, using nothing but the operator + for concatenation. In order to have parameter values as part of the texts the special substitution syntax is preferable (see section 18.7.5.5 and section 18.3.2.4), and translators need to be aware of these substrings in order to make good translations.

[*Example: Consider:*

```
annotation(..., Text(string = "1st Frequency: %f1"),
                Text(string = "2nd Frequency: " + String(w2 / (2 * pi))), ...);
```

In this example only `"1st Frequency is %f1."` *can be translated; the second* `Text.string` *doesn't consist entirely of concatenated string literals, and is hence completely excluded from translation.*]

The files to support translation must be provided along with the library. They must be stored in the resources directory modelica://*LibraryName*/Resources/Language/.

Two kind of files in Drepper, Meyering, Pinard, and Haible (2020) format have to be provided:

1. Template file *LibraryName*.pot (Portable Object Template), one file per library which is stored as the resource modelica://*LibraryName*/Resources/Language/*LibraryName*.pot. It describes all translatable strings in the library, but does not contain any translations. The pattern *LibraryName* denotes the toplevel class name of the library.

2. One file for each supported language with the name *LibraryName.language*.po (Portable Object), as the resource modelica://*LibraryName*/Resources/Language/*LibraryName.language*.po. This file is a copy of the associated template file, but extended with the translations in the specified language. The pattern *language* stands for the ISO 639-1 language code, e.g., de or sv.

The detailed format of these files is described in Drepper, Meyering, Pinard, and Haible (2020). Use of translation files in other formats (including the binary MO file format) is not standardized in Modelica. For Modelica translations, only the keywords `msgctxt`, `msgid` and `msgstr` are used, meaning that a translation entry looks like this:

```
#: filename:lineNumber
#, no-c-format
msgctxt messageContext
msgid messageIdentifier
msgstr messageTranslation
```

The restriction to a few keywords makes it easier for tools to support the format without relying on the implementation from Drepper, Meyering, Pinard, and Haible (2020).

The use of `no-c-format` ensures that translation tools will not parse `"%class"` as the format specifier `%c` followed by *lass*.

[*In the remainder of this section, several facts about the gettext specification are interleaved non-normatively for easy access to some of the gettext basics. Always refer to the external gettext specification for full detail or in case of doubt.*

All text strings are in double quotes and encoded with UTF-8 characters. Comments start with an `#` *and are continued until the end of line. Spaces outside strings are ignored and used as separators.*

The files consist of a header and a body. The header is marked with an empty `msgid` *and looks like this:*

```
# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR THE PACKAGE'S COPYRIGHT HOLDER
# This file is distributed under the same license as the PACKAGE
   package.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2019-03-15 10:52+0100\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"Language: \n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"
```

All general terms in the header should be replaced by specific information.

Following the header, there is one translation entry for each string to be translated. It can start with an optional comment describing the location (file name and line number) of the text to translate. Multiple occurences of the same string can be listed here, separated by space.]

The *messageContext* following the keyword `msgctxt` shall be the full name of the Modelica class (e.g., `"Modelica.Blocks.Math.Sin"`) where the text appears. Short class definitions do not appear here. Texts in such classes belong to the enclosing full class definition.

The text string which shall be translated is used as *messageIdentifier* (following the `msgid` keyword), and shall contain the original string from the Modelica code. Note that if a `msgid` string is given more than once in the same context, all occurrences are translated with the same (last) translation!

[*The messageTranslation (following the keyword* `msgstr`*) is the translation of messageIdentifier and is typically edited using special tools for translators. In the template file this string is empty by definition. If this is empty in a language specific file the messageIdentifier may be used instead.*]

[*Since in Modelica control sequences also start with a backslash and another backslash is used to use sequences literally or to hide double quotes, no change is required here. But Modelica allows strings to go over more than one line, gettext does not. Therefore, line breaks in Modelica must be encoded with* `"\n"` *for gettext.*

In order to avoid long lines in the translated strings (following `msgid` *or* `msgstr`*), strings may be split into smaller parts given on consecutive lines. E.g., the Modelica description*

```
"A
B\"C" + "D\nE"
```

evaluates to:

A
B"CD
E

which in the translation entry looks like:

```
msgid ""
"A\n"
"B\"CD\n"
"E"
```

]

[*Example: Consider a simple sine-source:*

```
within MyPackage.Sources;
model Sine "Sine"
  parameter Frequency f=2 "Frequency";
  RealOutput y=sin(2*pi*f*time); // Relying on imported types
  /* Could add details. Note that this is not translated */
  annotation(Icon(graphics={Text(extent={{0,0},{40,40}},
                        textString="Frequency: %f")}));
end Sine;
```

The entries for translating this model into Swedish could look like this:

```
#: MyPackage/Sources/package.mo:10126
#, no-c-format
msgctxt "MyPackage.Sources.Sine"
msgid "Sine"
msgstr "Sinus"
#: MyPackage/Sources/package.mo:10127
#, no-c-format
msgctxt "MyPackage.Sources.Sine"
msgid "Frequency"
msgstr "Frekvens"
#: MyPackage/Sources/package.mo:10131
#, no-c-format
msgctxt "MyPackage.Sources.Sine"
msgid "Frequency: %f"
msgstr "Frekvens: %f"
```

]

[*To support the translation of these strings a number of free and commercial tools exist in context of GNU gettext.*]

Chapter 14

Overloaded Operators

A Modelica **operator record** can overload the behavior for operations such as constructing, adding, multiplying etc.

The overloading is defined in such a way that ambiguities are not allowed and give an error. Furthermore, it is sufficient to define overloading for scalars. Some overloaded array operations are automatically deduced from the overloaded scalar operations (see item 4 and item 3 in the lists below), and others can be defined independently of the corresponding scalar operations.

14.1 Overview of Overloaded Operators

In an **operator record** the definition of operations are done using the specialized class **operator** (a specialized class similar to **package**, see section 4.7) followed by the name of the operation. Each **operator** class is comprised of functions implementing different variants of the operation for the **operator record** class in which the definition resides.

- Overloaded constructors, see section 14.3:
  `'constructor'`, `'0'`
- Overloaded string conversions, see section 14.4:
  `'String'`
- Overloaded binary operations, see section 14.5:
  `'+'`, `'-'` (subtraction), `'*'`, `'/'`, `'^'`, `'=='`, `'<='`, `'>'`, `'<'`, `'>='`, `'<='`, `'and'`, `'or'`
- Overloaded unary operations, see section 14.6:
  `'-'` (negation), `'not'`

The functions defined in the operator-class must take at least one component of the record class as input, except for the constructor-functions which instead must return one component of the record class. All of the functions shall return exactly one output.

The functions can be either called as defined in this section, or they can be called directly using the hierarchical name. The operator or operator function must be encapsulated; this allows direct calls of the functions and prohibits the functions from using the elements of operator record class.

The **operator record** may also contain additional functions, and declarations of components of the record. It is not legal to extend from an **operator record**, except as a short class definition modifying the default attributes for the component elements directly inside the operator record.

If an operator record was derived by a short class definition, the overloaded operators of this operator record are the operators that are defined in its base class, for subtyping see chapter 6.

The precedence and associativity of the overloaded operators is identical to the one defined in table 3.1 in section 3.2.

[*Note, the operator overloading as defined in this section is only a short hand notation for function calls.*]

14.2 Matching Function

All functions defined inside the **operator** class must return one output (based on the restriction above), and may include functions with optional arguments, i.e., functions of the form

```
function f
  input A1 u1;
  ...
  input Am um := am;
  ...
  input An un;
  output B y;
algorithm
  ...
end f;
```

The vector P indicates whether argument m of `f` has a default value (`true` for default value, `false` otherwise). A call `f(`a_1`, `a_2`, ..., `a_k`, `b_1` = `w_1`, ..., `b_p` = `w_p`)` with distinct names b_j is a valid match for the function `f`, provided (treating `Integer` and `Real` as the same type)

- A_i = typeOf(a_i) for $1 \le i \le k$,
- the names $b_j = u_{Q_j}$, $Q_j > k$, A_{Q_j} = typeOf(w_j) for $1 \le j \le p$, and
- if the union of $\{i : 1 \le i \le k\}$, $\{Q_j : 1 \le j \le p\}$, and $\{m : P_m$ is `true` and $1 \le m \le n\}$ is the set $\{i : 1 \le i \le n\}$.

[*This corresponds to the normal treatment of function calls with named arguments, requiring that all inputs have some value given by a positional argument, named argument, or a default value (and that positional and named arguments do not overlap). Note, that this only defines a valid call, but does not explicitly define the set of domains.*]

14.3 Overloaded Constructors

Let `C` denote an operator record class and consider an expression `C(`A_1`, `a_2`, ..., `a_k`, `b_1`=`w_1`, ..., `b_p`=`w_p`)`.

1. If there exists a unique function f in `C.'constructor'` such that (A_1, a_2, ..., a_k, b_1=w_1, ..., b_p=w_p) is a valid match for the function f, then `C(`A_1`, `a_2`, ..., `a_k`, `b_1`=`w_1`, ..., `b_p`=`w_p`)` is resolved to `C.'constructor'.`f`(`A_1`, `a_2`, ..., `a_k`, `b_1`=`w_1`, ..., `b_p`=`w_p`)`.
2. If there is no operator `C.'constructor'` the automatically generated record constructor is called.
3. Otherwise the expression is erroneous.

Restrictions:

- The operator `C.'constructor'` shall only contain functions that declare one output component, which shall be of the operator record class `C`.
- For an operator record class there shall not exist any potential call that lead to multiple matches in item 1 above.

  [*How to verify this is not specified.*]
- For a pair of operator record classes `C` and `D` and components `c` and `d` of these classes, respectively, at most one of `C.'constructor'(d)` and `D.'constructor'(c)` shall be legal.

  [*Hence, one of the two definitions must be removed.*]

[*By the last restriction the following problem for binary operators is avoided:*

Assume there are two operator record classes `C` *and* `D` *that both have a constructor from* `Real`*. If we want to extend* `c + c` *and* `d + d` *to support mixed operations, one variant would be to define* `c + d` *and* `d + c`*; but then* `c + 2` *becomes ambiguous (since it is not clear which instance should be converted to). Without mixed operations expressions such as* `c + d` *are only ambiguous if both conversion from* `C` *to* `D` *and back from* `D` *to* `C` *are both available, and this possibility is not allowed by the restriction above.*]

Additionally there is an operator `'0'` defining the zero-value which can also be used to construct an element. The operator `'0'` for an operator record `C` can contain only one function, having zero inputs and one output of type `C` (the called function is therefore unambiguous). It should return the identity element of addition, and is used for generating flow-equations for `connect`-equations and zero elements for matrix multiplication.

14.4 Overloaded String Conversions

Consider an expression `String(`A_1`, `a_2`, ..., `a_k`, `b_1`=`w_1`, ..., `b_p`=`w_p`)`, $k \geq 1$ where A_1 is an element of class `A`.

1. If `A` is a predefined type except `String` (i.e., `Boolean`, `Integer`, `Real` or an enumeration), or derived from such a type, then the corresponding built-in operation is performed.
2. If `A` is an operator record class and there exists a unique function f in `A.'String'` such that `A.'String'.`f`(`A_1`, `a_2`, ..., `a_k`, `b_1`=`w_1`, ..., `b_p`=`w_p`)` is a valid match for f, then `String(`A_1`, `a_2`, ..., `a_k`, `b_1`=`w_1`, ..., `b_p`=`w_p`)` is evaluated to `A.'String'.`f`(`A_1`, `a_2`, ..., `a_k`, `b_1`=`w_1`, ..., `b_p`=`w_p`)`.
3. Otherwise the expression is erroneous.

Restrictions:

- The operator `A.'String'` shall only contain functions that declare one output component, which shall be of the `String` type, and the first input argument shall be of the operator record class `A`.
- For an operator record class there shall not exist any call that lead to multiple matches in item 2 above.

  [*How to verify this is not specified.*]

14.5 Overloaded Binary Operations

Let X denote a binary operator and consider an expression `a` X `b` where `a` is an instance or array of instances of class `A` and `b` is an instance or array of instances of class `B`.

1. If `A` and `B` are predefined types of such, then the corresponding built-in operation is performed.
2. Otherwise, if there exists *exactly one* function f in the union of `A.`X and `B.`X such that f`(a, b)` is a valid match for the function f, then `a` X `b` is evaluated using this function. It is an error, if multiple functions match. If `A` is not an operator record class, `A.`X is seen as the empty set, and similarly for `B`.

   [*Having a union of the operators ensures that if* `A` *and* `B` *are the same, each function only appears once.*]

   Note that if the operations take array arguments, they will in this step only match if the number of dimensions match.
3. Otherwise, consider the set given by f in `A.`X and an operator record class `C` (different from `B`) with a constructor, g, such that `C.'constructor'.`g`(b)` is a valid match, and `f(a, C.'constructor'.`g`(b))` is a valid match; and another set given by f in `B.`X and an operator record class `D` (different from `A`) with a constructor, h, such that `D.'constructor'.`h`(a)` is a valid match and f`(D.'constructor'.`h`(a), b)` is a valid match. If the sum of the sizes of these sets is one this gives the unique match. If the sum of the sizes is larger than one it is an error. Note that if the operations take array arguments, they will in this step only match if the number of dimensions match.

   [*Informally, this means: If there is no direct match of* `a` X `b`*, then it is tried to find a direct match by automatic type casts of* `a` *or* `b`*, by converting either* `a` *or* `b` *to the needed type using an appropriate constructor function from one of the operator record classes used as arguments of the overloaded* `op` *functions. Example using the* `Complex`*-definition below:*

```
Real a;
Complex b;
Complex c = a * b; // interpreted as:
// Complex.'*'.multiply(Complex.'constructor'.fromReal(a), b);
```

]

4. Otherwise, if `a` or `b` is an array expression, then the expression is conceptually evaluated according to the rules of section 10.6 with the following exceptions concerning section 10.6.4:

   a. *vector* `*` *vector* is not automatically defined based on the scalar multiplication.

      [*The scalar product of table 10.10 does not generalize to the expected linear and conjugate linear scalar product of complex numbers. It is possible to define a specific product function taking two array arguments handling this case.*]

   b. *vector* `*` *matrix* is not automatically defined based on the scalar multiplication.

      [*The corresponding definition of table 10.10 does not generalize to complex numbers in the expected way. It is possible to define a specific product function taking two array arguments handling this case.*]

   c. If the inner dimension for *matrix* `*` *vector* or *matrix* `*` *matrix* is zero, this uses the overloaded `'0'` operator of the result array element type. If the operator `'0'` is not defined for that class it is an error if the inner dimension is zero.

   [*For array multiplication it is assumed that the scalar elements form a non-commutative ring that does not necessarily have a multiplicative identity.*]

5. Otherwise the expression is erroneous.

For an element-wise operator, `a .op b`, items 1, 4 and 5 are used; e.g., the operator `.+` will always be defined in terms of `'+'`.

Restrictions:

- A function is allowed for a binary operator if and only if it has at least two inputs; at least one of which is of the operator record class, and the first two inputs shall not have default values, and all inputs after the first two must have default values.
- For an operator record class there shall not exist any (potential) call that lead to multiple matches in item 2 above.

14.6 Overloaded Unary Operations

Let X denote a unary operator and consider an expression X `a` where `a` is an instance or array of instances of class `A`. Then X `a` is evaluated in the following way.

1. If `A` is a predefined type, then the corresponding built-in operation is performed.
2. If `A` is an operator record class and there exists a unique function f in `A.`X such that `A.`X`.`f`(a)` is a valid match, then X `a` is evaluated to `A.`X`.`f`(a)`. It is an error, if there are multiple valid matches. Note that if the operations take array arguments, they will in this step only match if the number of dimensions match.
3. Otherwise, if `a` is an array expression, then the expression is conceptually evaluated according to the rules of section 10.6.
4. Otherwise the expression is erroneous.

Restrictions:

- A function is allowed for a unary operator if and only if it has least one input; and the first input is of the record type (or suitable arrays of such) and does not have a default value, and all inputs after the first one must have default values.

- For an operator record class there shall not exist any (potential) call that lead to multiple matches in item 2 above.
- A binary and/or unary operator-class may only contain functions that are allowed for this binary and/or unary operator-class; and in case of `'-'` it is the union of these sets, since it may define both a unary (negation) and binary (subtraction) operator.

14.7 Example of Overloading for Complex Numbers

[*Example: The rules in the previous subsections are demonstrated at hand of a record class to work conveniently with complex numbers:*

```
operator record Complex "Record defining a Complex number"
  Real re "Real part of complex number";
  Real im "Imaginary part of complex number";
  encapsulated operator 'constructor'
    import Complex;
    function fromReal
      input Real re;
      input Real im := 0;
      output Complex result(re = re, im = im);
    algorithm
      annotation(Inline = true);
    end fromReal;
  end 'constructor';

  encapsulated operator function '+' // short hand notation, see section 4.7
    import Complex;
    input Complex c1;
    input Complex c2;
    output Complex result "= c1 + c2";
  algorithm
    result := Complex(c1.re + c2.re, c1.im + c2.im);
    annotation(Inline = true);
  end '+';

  encapsulated operator '-'
    import Complex;
    function negate
      input Complex c;
      output Complex result "= - c";
    algorithm
      result := Complex(-c.re, -c.im);
      annotation(Inline = true);
    end negate;

    function subtract
      input Complex c1;
      input Complex c2;
      output Complex result "= c1 - c2";
    algorithm
      result := Complex(c1.re - c2.re, c1.im - c2.im);
      annotation(Inline = true);
    end subtract;
  end '-';

  encapsulated operator function '*'
    import Complex;
    input Complex c1;
    input Complex c2;
    output Complex result "= c1 * c2";
  algorithm
    result :=
```

```
      Complex(c1.re * c2.re - c1.im * c2.im, c1.re * c2.im + c1.im * c2.re);
    annotation(Inline = true);
  end '*';

  encapsulated operator function '/'
    import Complex; input Complex c1;
    input Complex c2;
    output Complex result "= c1 / c2";
  algorithm
    result :=
      Complex((c1.re*c2.re + c1.im*c2.im) / (c2.re^2 + c2.im^2),
              (-c1.re*c2.im + c1.im*c2.re) / (c2.re^2 + c2.im^2));
    annotation(Inline = true);
  end '/';

  encapsulated operator function '=='
    import Complex;
    input Complex c1;
    input Complex c2;
    output Boolean result "= c1 == c2";
  algorithm
    result := c1.re == c2.re and c1.im == c2.im;
    annotation(Inline = true);
  end '==';

  encapsulated operator function 'String'
    import Complex;
    input Complex c;
    input String name := "j"
      "Name of variable representing sqrt(-1) in the string";
    input Integer significantDigits = 6
      "Number of significant digits to be shown";
    output String s;
  algorithm
    s := String(c.re, significantDigits = significantDigits);
    if c.im <> 0 then
      s := if c.im > 0 then s + " + " else s + " - ";
      s := s + String(abs(c.im), significantDigits = significantDigits) + name;
    end if;
  end 'String';

  encapsulated function j
    import Complex;
    output Complex c;
  algorithm
    c := Complex(0, 1);
    annotation(Inline = true);
  end j;

  encapsulated operator function '0'
    import Complex;
    output Complex c;
  algorithm
    c := Complex(0, 0);
    annotation(Inline = true);
  end '0';
end Complex;

function eigenValues
  input Real A [:,:];
  output Complex ev[size(A, 1)];
  protected
  Integer nx = size(A, 1);
```

```
  Real eval[nx, 2];
  Integer i;
algorithm
  eval := Modelica.Math.Matrices.eigenValues(A);
  for i in 1 : nx loop
    ev[i] := Complex(eval[i, 1], eval[i, 2]);
  end for;
end eigenValues;

// Usage of Complex number above:
  Complex j = Complex.j();
  Complex c1 = 2 + 3 * j;
  Complex c2 = 3 + 4 * j;
  Complex c3 = c1 + c2;
  Complex c4[:] = eigenValues([1, 2; -3, 4]);
algorithm
  Modelica.Utilities.Streams.print("c4 = " + String(c4));
  // results in output:
  // c4 = {2.5 + 1.93649j, 2.5 - 1.93649j}
```

How overloaded operators can be symbolically processed. Example:

```
Real a;
Complex b;
Complex c = a + b;
```

Due to inlining of functions, the equation for `c` *is transformed to:*

```
c = Complex.'+'.add(Complex.'constructor'.fromReal(a), b);
  = Complex.'+'.add(Complex(re = a, im = 0), b)
  = Complex(re = a + b.re, im = b.im);
```

or

```
c.re = a + b.re;
c.im = b.im;
```

These equations can be symbolically processed as other equations.

Complex can be used in a connector:

```
  operator record ComplexVoltage = Complex(re(unit = "V"), im(unit = "V"));
  operator record ComplexCurrent = Complex(re(unit = "A"), im(unit = "A"));

  connector ComplexPin
    ComplexVoltage v;
    flow ComplexCurrent i;
  end ComplexPin;

  ComplexPin p1, p2, p3;
equation
  connect(p1, p2);
  connect(p1, p3);
```

The two `connect`*-equations result in the following connection equations:*

```
p1.v = p2.v;
p1.v = p3.v;
p1.i + p2.i + p3.i = Complex.'0'();
// Complex.'+'(p1.i, Complex.'+'(p2.i, p3.i)) = Complex.'0'();
```

The restrictions on extends are intended to avoid combining two variants inheriting from the same operator record, but with possibly different operations; thus `ComplexVoltage` *and* `ComplexCurrent` *still use the operations from* `Complex`*. The restriction that it is not legal to extend from any of its enclosing scopes implies that:*

```
package A
  extends Icon; // Ok
  operator record B ... end B;
end A;

package A2
  extends A(...); // Not legal
end A2;

package A3 = A(...); // Not legal
```

]

Chapter 15

Stream Connectors

The two basic variable types in a connector – *potential* (or *across*) variable and *flow* (or *through*) variable – are not sufficient to describe in a numerically sound way the bi-directional flow of matter with convective transport of specific quantities, such as specific enthalpy and chemical composition. The values of these specific quantities are determined from the upstream side of the flow, i.e., they depend on the flow direction. When using across and through variables, the corresponding models would include nonlinear systems of equations with `Boolean` unknowns for the flow directions and singularities around zero flow. Such equation systems cannot be solved reliably in general. The model formulations can be simplified when formulating two different balance equations for the two possible flow directions. This is not possible with across and through variables though.

This fundamental problem is addressed in Modelica by introducing a third type of connector variable, called *stream variable*, declared with the prefix `stream`. A stream variable describes a quantity that is carried by a flow variable, i.e., a purely convective transport phenomenon. The value of the stream variable is the specific property inside the component close to the boundary, assuming that matter flows out of the component into the connection point. In other words, it is the value the carried quantity would have if the fluid was flowing out of the connector, irrespective of the actual flow direction.

The rationale of the definition and typical use cases are described in appendix C.

15.1 Definition of Stream Connectors

If at least one variable in a connector has the `stream` prefix, the connector is called *stream connector* and the corresponding variable is called *stream variable*. The following definitions hold:

- The `stream` prefix can only be used in a connector declaration.
- A stream connector must have exactly one variable with the `flow` prefix. That variable shall be a scalar that is a subtype of `Real`.

  [*The idea is that all stream variables of a connector are associated with this flow variable.*]
- For every outside connector (see section 9.1.2), one equation is generated for every variable with the `stream` prefix (to describe the propagation of the stream variable along a model hierarchy). For the exact definition, see the end of section 15.2.
- For inside connectors (see section 9.1.2), variables with the `stream` prefix do not lead to connection equations.
- Connection equations with stream variables are generated in a model when using `inStream` or `actualStream`, see section 15.2 and section 15.3.

[*Example:*

```
connector FluidPort
  replaceable package Medium =
    Modelica.Media.Interfaces.PartialMedium;
  Medium.AbsolutePressure p "Pressure in connection point";
```

```
  flow Medium.MassFlowRate m_flow "> 0, if flow into component";
  stream Medium.SpecificEnthalpy h_outflow "h close to port if m_flow < 0";
  stream Medium.MassFraction X_outflow[Medium.nX] "X close to port if m_flow <
    0";
end FluidPort;
```

`FluidPort` *is a stream connector, because some connector variables have the* `stream` *prefix. The Medium definition and the stream variables are associated with the only flow variable (*`m_flow`*) that defines a fluid stream. The Medium and the stream variables are transported with this flow variable. The stream variables* `h_outflow` *and* `X_outflow` *are the stream properties inside the component close to the boundary, when fluid flows out of the component into the connection point. The stream properties for the other flow direction can be inquired with the built-in* `inStream`*. The value of the stream variable corresponding to the actual flow direction can be inquired through the built-in* `actualStream`*, see section 15.3.*]

15.2 inStream and Connection Equations

In combination with the stream variables of a connector, `inStream` is designed to describe in a numerically reliable way the bi-directional transport of specific quantities carried by a flow of matter.

`inStream(v)` is only allowed on stream variables `v` and is informally the value the stream variable has, assuming that the flow is from the connection point into the component. This value is computed from the *stream connection equations* of the flow variables and of the stream variables.

For the following definition it is assumed that N inside connectors m_j`.c` $(j = 1, 2, \ldots, N)$ and M outside connectors c_k $(k = 1, 2, \ldots, M)$ belonging to the same connection set (see definition in section 9.1.2) are connected together and a stream variable `h_outflow` is associated with a flow variable `m_flow` in connector `c`.

```
connector FluidPort
  ...
  flow Real m_flow "Flow of matter; m_flow > 0 if flow into component";
  stream Real h_outflow "Specific variable in component if m_flow < 0"
end FluidPort

model FluidSystem
  ...
  FluidComponent m1, m2, ..., mN;
  FluidPort c1, c2, ..., cM;
equation
  connect(m1.c, m2.c);
  connect(m1.c, m3.c);
  ...
  connect(m1.c, mN.c);
  connect(m1.c, c1);
  connect(m1.c, c2);
  ...
  connect(m1.c, cM);
  ...
end FluidSystem;
```

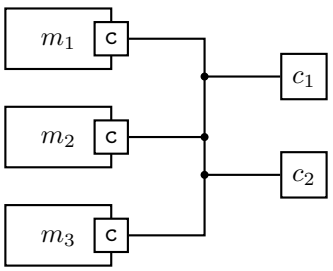


Figure 15.1: *Examplary* `FluidSystem` *with* $N = 3$ *and* $M = 2$.

[*The connection set represents an infinitesimally small control volume, for which the stream connection equations are equivalent to the conservation equations for mass and energy.*]

With these prerequisites, the semantics of the expression `inStream(`m_i`.c.h_outflow)` is given implicitly by defining an additional variable $h\_mix\_in_i$, and by adding to the model the conservation equations for mass and energy corresponding to the infinitesimally small volume spanning the connection set. The connection equation for the flow variables has already been added to the system according to the connection semantics of flow variables defined in section 9.2.

```
// Standard connection equation for flow variables
0 = sum(mj.c.m_flow for j in 1:N) + sum(-ck.m_flow for k in 1:M);
```

Whenever `inStream` is applied to a stream variable of an inside connector, the balance equation of the transported property must be added under the assumption of flow going into the connector

```
// Implicit definition of inStream applied to inside connector i
0 =
  sum(mj.c.m_flow *
      (if mj.c.m_flow > 0 or j==i then h_mix_ini else mj.c.h_outflow)
    for j in 1:N) +
  sum(-ck.m_flow *
      (if -ck.m_flow > 0 then h_mix_ini else inStream(ck.h_outflow)
    for k in 1:M);
inStream(mi.c.h_outflow) = h_mix_ini;
```

Note that the result of `inStream(`m_i`.c.h_outflow)` is different for each port i, because the assumption of flow entering the port is different for each of them.

Additional equations need to be generated for the stream variables of outside connectors.

```
// Additional connection equations for outside connectors
for q in 1:M loop
  0 =
    sum(mj.c.m_flow *
        (if mj.c.m_flow > 0 then h_mix_outq else mj.c.h_outflow)
      for j in 1:N) +
    sum(-ck.m_flow *
        (if -ck.m_flow > 0 or k==q then h_mix_outq else inStream(ck.h_outflow))
      for k in 1:M);
  cq.h_outflow = h_mix_outq;
end for;
```

Neglecting zero flow conditions, the solution of the above-defined stream connection equations for `inStream` values of inside connectors and outflow stream variables of outside connectors is (for a derivation, see appendix C):

```
inStream(mi.c.h_outflow) :=
  (sum(max(-mj.c.m_flow,0)*mj.c.h_outflow for j in cat(1, 1:i-1, i+1:N) +
   sum(max( ck.m_flow,0)*inStream(ck.h_outflow) for k in 1:M))
  /
  (sum(max(-mj.c.m_flow,0) for j in cat(1, 1:i-1, i+1:N) +
   sum(max( ck.m_flow ,0) for k in 1:M));

// Additional equations to be generated for outside connectors q
for q in 1:M loop
  cq.h_outflow :=
    (sum(max(-mj.c.m_flow,0)*mj.c.h_outflow for j in 1:N) +
     sum(max( ck.m_flow,0)*inStream(ck.h_outflow) for k in cat(1, 1:q-1, q+1:M))
    /
    (sum(max(-mj.c.m_flow,0) for j in 1:N) +
     sum(max( ck.m_flow ,0) for k in cat(1, 1:q-1, q+1:M)));
end for;
```

[*Note, that* `inStream(`c_k`.h_outflow)` *is computed from the connection set that is present one hierarchical level above. At this higher level* c_k`.h_outflow` *is no longer an outside connector, but an inside connector*

and then the formula from above for inside connectors can be used to compute it.]

If the argument of `inStream` is an array, the implicit equation system holds elementwise, i.e., `inStream` is vectorizable.

The stream connection equations have singularities and/or multiple solutions if one or more of the flow variables become zero. When all the flows are zero, a singularity is always present, so it is necessary to approximate the solution in an open neighbourhood of that point.

[*For example, assume that* m_j`.c.m_flow` = c_k`.m_flow = 0`, *then all equations above are identically fulfilled and* `inStream` *can have any value.*]

However, specific optimizations may be applied to avoid the regularization if the flow through one port is zero or non-negative, see appendix C. It is required that `inStream` is appropriately approximated when regularization is needed and the approximation must fulfill the following requirements:

1. `inStream(`m_i`.c.h_outflow)` and `inStream(`c_k`.h_outflow)` must be unique with respect to all values of the flow and stream variables in the connection set, and must have a continuous dependency on them.
2. Every solution of the implicit equation system above must fulfill the equation system identically (upto the usual numerical accuracy), provided the absolute value of every flow variable in the connection set is greater than a small value ($|m_i$`.c.m_flow`$| >$ `eps` for $1 \leq i \leq N$ and $|c_i$`.m_flow`$| >$ `eps` for $1 \leq i \leq M$).

[*Based on the above requirements, the following implementation is recommended:*

- $N = 1$, $M = 0$*:*

```
inStream(m1.c.h_outflow) = m1.c.h_outflow;
```

- $N = 2$, $M = 0$*:*

```
inStream(m1.c.h_outflow) = m2.c.h_outflow;
inStream(m2.c.h_outflow) = m1.c.h_outflow;
```

- $N = 1$, $M = 1$*:*

```
inStream(m1.c.h_outflow) = inStream(c1.h_outflow);
// Additional equation to be generated
c1.h_outflow = m1.c.h_outflow;
```

- $N = 0$, $M = 2$*:*

```
// Additional equation to be generated
c1.h_outflow = inStream(c2.h_outflow);
c2.h_outflow = inStream(c1.h_outflow);
```

- *All other cases:*

```
if mj.c.m_flow.min >= 0  for all j = 1:N with j <> i  and
   ck.m_flow.max <= 0 for all k = 1:M
then
  inStream(mi.c.h_outflow) = mi.c.h_outflow;
else
  si = sum (max(-mj.c.m_flow,0) for j in cat(1,1:i-1, i+1:N) +
       sum(max( ck.m_flow ,0) for k  in 1:M);
  inStream(mi.c.h_outflow) =
     (sum(positiveMax(-mj.c.m_flow,si)*mj.c.h_outflow)
    +  sum(positiveMax(ck.m_flow,s_i)*inStream(ck.h_outflow)))/
   (sum(positiveMax(-mj.c.m_flow,s_i))
      +  sum(positiveMax(ck.m_flow,s_i)))
                for j in 1:N and i <> j and mj.c.m_flow.min < 0,
                for k in 1:M and ck.m_flow.max > 0
// Additional equations to be generated
for q in 1:M loop
  if mj.c.m_flow.min >= 0 for all j = 1:N and
```

```
    c_k.m_flow.max <= 0 for all k = 1:M and k <> q
  then
    c_q.h_outflow = 0;
  else
    s_q = (sum(max(-m_j.c.m_flow,0) for j in  1:N) +
                  sum(max( c_k.m_flow ,0) for k in cat(1,1:q-1, q+1:M)));
    c_q.h_outflow = (sum(positiveMax(-m_j.c.m_flow,s_q)*m_j.c.h_outflow) +
                sum(positiveMax(c_k.m_flow,s_q)* inStream(c_k.h_outflow)))/
               (sum(positiveMax(-m_j.c.m_flow,s_q)) +
                sum(positiveMax(c_k.m_flow,s_q)))
    for j in 1:N and m_j.c.m_flow.min < 0,
                 for k in 1:M and k <> q and c_k.m_flow.max > 0
end for;
```

The operator positiveMax(-m_j.c.m_flow, s_i) *should be such that:*

- positiveMax(-m_j.c.m_flow, s_i) = -m_j.c.m_flow *if* $-m_j$.c.m_flow $> eps1_j \geq 0$, *where* $eps1_j$ *are small flows, compared to typical problem-specific values,*
- *all denominators should be greater than* $eps2 > 0$, *where* $eps2$ *is also a small flow, compared to typical problem-specific values.*

Trivial implementation of positiveMax *guarantees continuity of* inStream*:*

```
postiveMax(-m_j.c.m_flow, s_i) = max(-m_j.c.m_flow, eps1); // so s_i is not needed
```

More sophisticated implementation, with smooth approximation, applied only when all flows are small:

```
// Define a "small number" eps (nominal(v) is the nominal value of v, see
    section 4.9.6)
eps := relativeTolerance*min(nominal(m_j.c.m_flow));

// Define a smooth curve, such that  alpha(s_i >= eps)=1 and alpha(s_i < 0)=0
alpha := smooth(1, if s_i > eps then 1
                   else if s_i > 0 then (s_i/eps)^2*(3-2* s_i/eps)
                   else 0);

// Define function positiveMax(v,s_i) as a linear combination of max (v,0)
// and of eps along alpha
positiveMax((-m_j.c.m_flow,s_i) := alpha*max(-m_j.c.m_flow,0) + (1-alpha)*eps;
```

The derivation of this implementation is discussed in appendix C. Note that in the cases $N = 1$, $M = 0$ *(unconnected port, physically corresponding to a plugged-up flange), and* $N = 2$, $M = 0$ *(one-to-one connection), the result of* inStream *is trivial and no non-linear equations are left in the model, despite the fact that the original definition equations are nonlinear.*

The following properties hold for this implementation:

- inStream *is continuous (and differentiable), provided that* m_j.c.h_outflow, m_j.c.m_flow, c_k.h_outflow, *and* c_k.m_flow *are continuous and differentiable.*
- *A division by zero can no longer occur (since* sum(positiveMax(-m_j.c.m_flow, s_i)) $\geq eps2 > 0$*), so the result is always well-defined.*
- *The balance equations are exactly fulfilled if the denominator is not close to zero (since the exact formula is used, if* sum(positiveMax(-m_j.c.m_flow, s_i)) $> eps$*).*
- *If all flows are zero,* inStream(m_i.c.h_outflow) = sum(m_j.c.h_outflow **for** $j \neq i$ and m_j.c.m_flow.min < 0) / N_p*, i.e., it is the mean value of all the* N_p *variables* m_j.c.h_outflow*, such that* $j \neq i$ *and* m_j.c.m_flow.min < 0*. This is a meaningful approximation, considering the physical diffusion effects that are relevant at small flow rates in a small connection volume (thermal conduction for enthalpy, mass diffusion for mass fractions).*

The value of relativeTolerance *should be larger than the relative tolerance of the nonlinear solver used to solve the implicit algebraic equations.*

As a final remark, further symbolic simplifications could be carried out by taking into account equations that affect the flows in the connection set (i.e., equivalent to m_j`.c.m_flow` $= 0$*, which then implies* m_j`.c.m_flow.min` ≥ 0*). This is interesting, e.g., in the case of a valve when the stem position is set identically to closed by its controller.*]

15.3 actualStream

`actualStream` is provided for convenience, in order to return the actual value of the stream variable, depending on the actual flow direction. The only argument of this built-in operator needs to be a reference to a stream variable. The operator is vectorizable, in the case of vector arguments. For the following definition it is assumed that an (inside or outside) connector `c` contains a stream variable `h_outflow` which is associated with a flow variable `m_flow` in the same connector `c`:

```
actualStream(c.h_outflow) =
  if c.m_flow > 0 then inStream(c.h_outflow) else c.h_outflow;
```

[`actualStream` *is typically used in two contexts:*

```
der(U) = c.m_flow * actualStream(c.h_outflow);  // (1) energy balance equation
h_c = actualStream(c.h);                        // (2) monitoring the enthalpy
    at port c
```

In the case of equation (1), although `actualStream` *is discontinuous, the product with the flow variable is not, because* `actualStream` *is discontinuous when the flow is zero by construction. Therefore, a tool might infer that the expression is* `smooth(0, ...)` *automatically, and decide whether or not to generate an event. If a user wants to avoid events entirely, he/she may enclose the right-hand side of (1) with* `noEvent`.

Equations like (2) might be used for monitoring purposes (e.g., plots), in order to inspect what the actual enthalpy of the fluid flowing through a port is. In this case, the user will probably want to see the change due to flow reversal at the exact instant, so an event should be generated. If the user doesn't bother, then he/she should enclose the right-hand side of (2) with `noEvent`. *Since the output of* `actualStream` *will be discontinuous, it should not be used by itself to model physical behaviour (e.g., to compute densities used in momentum balances) –* `inStream` *should be used for this purpose.* `actualStream` *should be used to model physical behaviour only when multiplied by the corresponding flow variable (like in the above energy balance equation), because this removes the discontinuity.*]

Chapter 16

Synchronous Language Elements

This chapter defines synchronous behavior suited for implementation of control systems. The synchronous behavior relies on an additional kind of discrete-time variables and equations, as well as an additional kind of **when**-clause. The benefits of synchronous behavior is that it allows a model to define large sampled data systems in a safe way, so that the translator can provide good a diagnostic in case of a modeling error.

The following small example shows the most important elements:

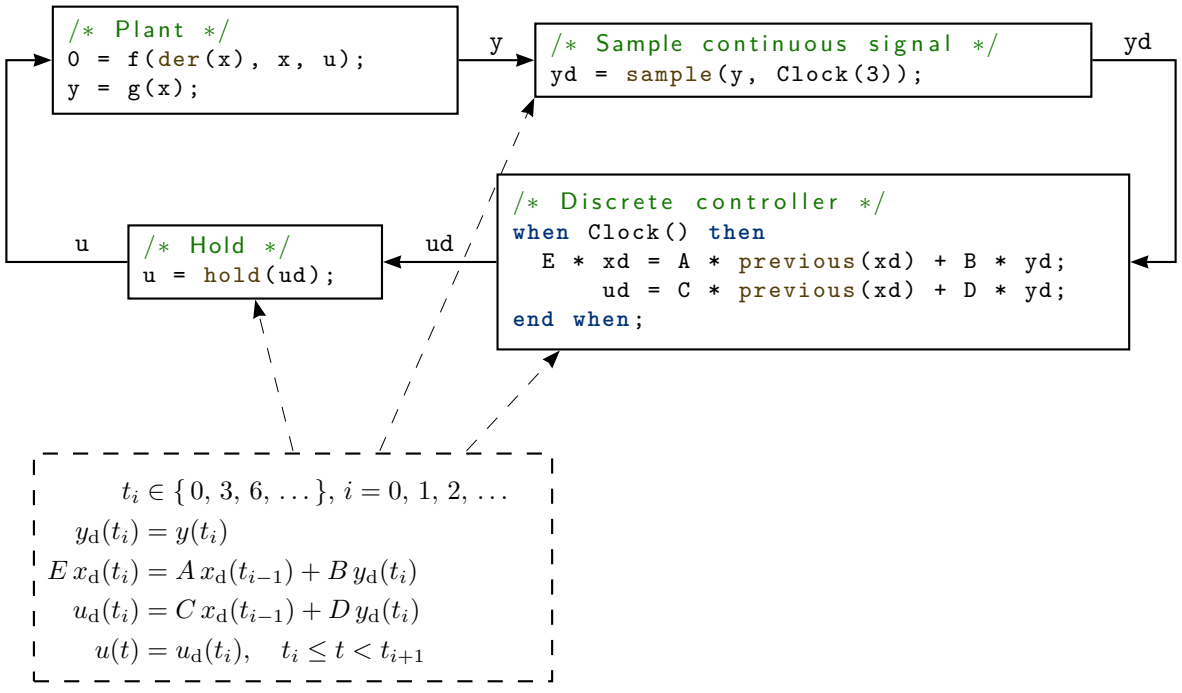


Figure 16.1: *A continuous plant and a sampled data controller connected together with sample and (zero-order) hold elements.*

- A periodic clock is defined with `Clock(3)`. The argument of `Clock` defines the sampling interval (for details see section 16.3).
- Clocked variables (such as **yd**, **xd**, **ud**) are associated uniquely with a clock and can only be directly accessed when the associated clock is active. Since all variables in a clocked equation must belong to the same clock, clocking errors can be detected at compile time. If variables from different clocks shall be used in an equation, explicit cast operators must be used, such as `sample` to convert from continuous-time to clocked discrete-time or `hold` to convert from clocked discrete-time to continuous-time.

- A continuous-time variable is sampled at a clock tick with `sample`. The operator returns the value of the continuous-time variable when the clock is active.
- When no argument is defined for `Clock`, the clock is deduced by clock inference.
- For a **when**-clause with an associated clock, all equations inside the **when**-clause are clocked with the given clock. All equations on an associated clock are treated together and in the same way regardless of whether they are inside a **when**-clause or not. This means that automatic sampling and hold of variables inside the **when**-clause does not apply (explicit sampling and hold is required) and that general equations can be used in such **when**-clauses (this is not allowed for **when**-clauses with `Boolean` conditions, that require a variable reference on the left-hand side of an equation).
- The **when**-clause in the controller could also be removed and the controller could just be defined by the equations:

```
/* Discrete controller */
E * xd = A * previous(xd) + B * yd;
    ud = C * previous(xd) + D * yd;
```

- `previous(xd)` returns the value of **xd** at the previous clock tick. At the first sample instant, the start value of **xd** is returned.
- A discrete-time signal (such as **ud**) is converted to a continuous-time signal with `hold`.
- If a variable belongs to a particular clock, then all other equations where this variable is used, with the exception of as argument to certain special operators, belong also to this clock, as well as all variables that are used in these equations. This property is used for clock inference and allows defining an associated clock only at a few places (above only in the sampler, whereas in the discrete controller and the hold the sampling period is inferred).
- The approach in this chapter is based on the clock calculus and inference system proposed by Colaço and Pouzet (2003) and implemented in Lucid Synchrone version 2 and 3 (Pouzet 2006). However, the Modelica approach also uses multi-rate periodic clocks based on rational arithmetic introduced by Forget, Boniol, Lesens, and Pagetti (2008), as an extension of the Lucid Synchrone semantics. These approaches belong to the class of synchronous languages (Benveniste, Caspi, Edwards, Halbwachs, Le Guernic, and Simone 2003).

16.1 Rationale for Clocked Semantics

[*Periodically sampled control systems could also be defined with standard* **when***-clauses, see section 8.3.5, and the* `sample` *operator, see section 3.7.5. For example:*

```
when sample(0, 3) then
  xd = A * pre(xd) + B * y;
  u  = C * pre(xd) + D * y;
end when;
```

Equations in a **when***-clause with a* `Boolean` *condition have the property that (a) variables on the left hand side of the equal sign are assigned a value when the when-condition becomes true and otherwise hold their value, (b) variables not assigned in the* **when***-clause are directly accessed (= automatic* `sample` *semantics), and (c) the variables assigned in the* **when***-clause can be directly accessed outside of the* **when***-clause (= automatic* `hold` *semantics).*

Using standard **when***-clauses works well for individual simple sampled blocks, but the synchronous approach using clocks and clocked equations provide the following benefits (especially for large sampled systems):*

1. *Possibility to detect inconsistent sampling rate, since clock partitioning (see section 16.7), replaces the automatic sample and hold semantics. Examples:*
   a. *If* **when***-clauses in different blocks should belong to the same controller part, but by accident different when-conditions are given, then this is accepted (no error is detected).*
   b. *If a sampled data library such as the Modelica_LinearSystems2.Contoller library is used, at every block the sampling of the block has to be defined as integer multiple of a base sampling*

rate. If several blocks should belong to the same controller part, and different integer multiples are given, then the translator has to accept this (no error is detected).

Note: Clocked systems can mix different sampling rates in well-defined ways when needed.

2. *Fewer initial conditions are needed, as only a subset of clocked variables need initial conditions – the clocked state variables (see section 16.4). For a standard* **when***-clause all variables assigned in a* **when***-clause must have an initial value because they might be used, before they are assigned a value the first time. As a result, all these variables are "discrete-time states" although in reality only a subset of them need an initial value.*

3. *More general equations can be used, compared to standard* **when***-clauses that require a restricted form of equations where the left hand side has to be a variable, in order to identify the variables that are assigned in the* **when***-clause. This restriction can be circumvented for standard* **when***-clauses, but is absent for clocked equations and make it more convenient to define nonlinear control algorithms.*

4. *Clocked equations allow clock inference, meaning that the sampling need only be given once for a sub-system. For a standard* **when***-clause the condition (sampling) must be explicitly propagated to all blocks, which is tedious and error prone for large systems.*

5. *Possible to use general continuous-time models in synchronous models (e.g., some advanced controllers use an inverse model of a plant in the feedforward path of the controller, see Thümmel, Looye, Kurze, Otter, and Bals (2005)). This powerful feature of Modelica to use a nonlinear plant model in a controller would require to export the continuous-time model with an embedded integration method and then import it in an environment where the rest of the controller is defined. With clocked equations, clocked controllers with continuous-time models can be directly defined in Modelica.*

6. *Clocked equations are straightforward to optimize because they are evaluated exactly once at each event instant. In contrast a standard* **when***-clause with* `sample` *conceptually requires several evaluations of the model (in some cases tools can optimize this to avoid unneeded evaluations). The problem for the standard* **when***-clause is that after* `v` *is changed,* `pre(v)` *shall be updated and the model re-evaluated, since the equations could depend on* `pre(v)`*. For clocked equations this iteration can be omitted since* `previous(v)` *can only occur in the clocked equations that are only run the first event iterations.*

7. *Clocked subsystems using arithmetic blocks are straightforward to optimize. When a standard math-block (e.g., addition) is part of a clocked sub-system it is automatically clocked and only evaluated when the clocked equations trigger. For standard* **when***-clauses one either needs a separate sampled math-block for each operation, or it will conceptually be evaluated all the time. However, tools may perform a similar optimization for standard* **when***-clauses and it is only relevant in large sampled systems.*

]

16.2 Definitions

In this section various terms are defined.

16.2.1 Clocks and Clocked Variables

In section 3.8.5 the term *discrete-time* Modelica expression and in section 3.8.6 the term *continuous-time* Modelica expression is defined. In this chapter, two additional kinds of discrete-time expressions/variables are defined that are associated to clocks and are therefore called *clocked discrete-time* expressions. The different kinds of discrete-time variables in Modelica are defined below.

Definition 16.1. ***Piecewise-constant variable.*** (See section 3.8.5.) Variables $m(t)$ of base type `Real`, `Integer`, `Boolean`, enumeration, and `String` that are *constant* inside each interval $t_i \leq t < t_{i+1}$ (i.e., piecewise constant continuous-time variables). In other words, $m(t)$ changes value only at events: $m(t) = m(t_i)$, for $t_i \leq t < t_{i+1}$. Such variables depend continuously on time and they are discrete-time variables. See figure 16.2. □

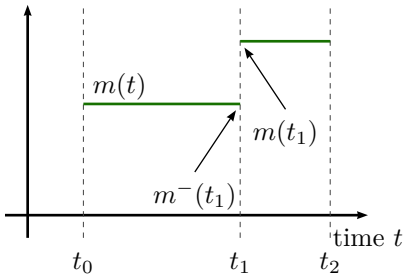


Figure 16.2: *A piecewise-constant variable.*

Definition 16.2. ***Clock variable.*** Clock variables $c(t_i)$ are of base type `Clock`. A clock is either defined by a constructor (such as `Clock(3)`) that defines when the clock ticks (is active) at a particular time instant, or it is defined with clock operators relatively to other clocks, see section 16.5.1. See figure 16.3. □

[*Example: Clock variables:*

```
Clock c1 = Clock(...);
Clock c2 = c1;
Clock c3 = subSample(c2, 4);
```

]

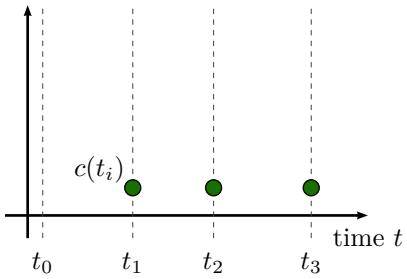


Figure 16.3: *A clock variable. The value of a clock variable is not defined – the plot marks only indicate* when *the clock is active.*

Definition 16.3. ***Clocked variable.*** The elements of clocked variables $r(t_i)$ are of base type `Real`, `Integer`, `Boolean`, enumeration, `String` that are associated uniquely with a clock $c(t_i)$. A clocked variable can only be directly accessed at the event instant where the associated clock is active. A constant and a parameter can always be used at a place where a clocked variable is required.

[*Note that clock variables are not included in this list. This implies that clock variables cannot be used where clocked variables are required.*]

At time instants where the associated clock is not active, the value of a clocked variable can be inquired by using an explicit cast operator, see below. In such a case `hold` semantics is used, in other words the value of the clocked variable from the last event instant is used. See figure 16.4. □

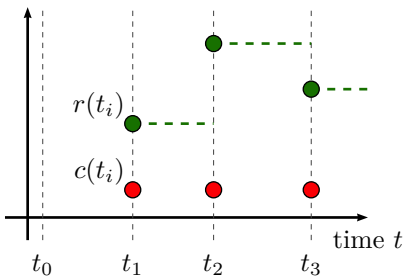


Figure 16.4: *A clocked variable. The* `hold` *extrapolation of the value at the last event instant is illustrated with dashed green lines.*

16.2.2 Base- and Sub-Partitions

There are two kinds of *clock partitions*:

Definition 16.4. ***Base-partition.*** A base-partition identifies a set of equations and a set of variables which must be executed together in one task. Different base-partitions can be associated to separate tasks for asynchronous execution. □

Definition 16.5. ***Sub-partition.*** A sub-partition identifies a subset of equations and a subset of variables of a base-partition which are partially synchronized with other sub-partitions of the same base-partition, i.e., synchronized when the ticks of the respective clocks are simultaneous. □

The terminology for the partitions is as follows:

- *Clocked base-partitions.*
  - *Discrete-time sub-partitions.*
  - *Discretized sub-partitions.*
- *Continuous-time base-partition.*

[*Note that the term* clock partition *refers to these partitions in general, whereas* clocked base-partition *is a specific kind of partition. Previously the discrete-time sub-partitions were called* clocked discrete-time *(*sub-clock *partition). Further, discretized sub-partitions were called* discretized continuous-time *(*sub-clock *partition). When emphasizing that the partitions are clock partitions, sub-partitions can still be referred to as* sub-clock partitions*; and similarly for base-partition.*]

16.2.3 Argument Restrictions (Component Expression)

The built-in operators (with function syntax) defined in the following sections have partially restrictions on their input arguments that are not present for Modelica functions. To define the restrictions, the following term is used.

Definition 16.6. ***Component expression.*** A component expression is a `component-reference` which is a valid expression, i.e., not referring to models or blocks with equations. In detail, it is an instance of a (a) base type, (b) derived type, (c) record, (d) an array of such an instance (a-c), (e) one or more elements of such an array (d) defined by index expressions which are evaluable (see below), or (f) an element of records.

[*The essential features are that one or several values are associated with the instance, that start values can be defined on these values, and that no equations are associated with the instance. A component expression can be constant or can vary with time.*]

□

In the following sections, when defining an operator with function calling syntax, there are some common restrictions being used for the input arguments (operands). For example, an input argument to the operator may be required to be a component expression (definition 16.6) or evaluable expression (section 3.8). To emphasize that there are no such restrictions, an input argument may be said to be just an *expression*.

[*The reason for restricting an input argument to be a component expression is that the start value of the input argument is returned before the first tick of the clock of the input argument and this is not possible for a general expression.*

The reason for restricting an input argument to be an evaluable expression is to ensure that clock analysis can be performed during translation. In cases when special handling of parameter expressions is specified, it is an indication that the values are not needed during translation.]

[*Example: The input argument to* `previous` *is restricted to be a component expression.*

```
Real u1;
Real u2[4];
Complex c;
Resistor R;
...
```

```
y1 = previous(u1);     // fine
y2 = previous(u2);     // fine
y3 = previous(u2[2]); // fine
y4 = previous(c.im);  // fine
y5 = previous(2 * u); // error (general expression, not component expression)
y6 = previous(R);      // error (component, not component expression)
```

]

[*Example: The named argument* `factor` *of* `subSample` *is restricted to be an evaluable expression.*

```
Real u;
parameter Real p=3;
...
y1 = subSample(u, factor = 3);          // fine (literal)
y2 = subSample(u, factor = 2 * p - 3); // fine (evaluable expression)
y3 = subSample(u, factor = 3 * u);      // error (general expression)
```

]

None of the operators defined in this chapter vectorize, but some can operate directly on array variables (including clocked array variables, but not clock array variables). They are not callable in functions.

16.3 Clock Constructors

The overloaded constructors listed below are available to generate clocks, and it is possible to call them with the specified named arguments, or with positional arguments (according to the order shown in the details after the table).

Expression	*Description*	*Details*
`Clock()`	Inferred clock	Operator 16.1
`Clock(intervalCounter, resolution)`	Rational interval clock	Operator 16.2
`Clock(interval)`	Real interval clock	Operator 16.3
`Clock(condition, startInterval)`	Event clock	Operator 16.4
`Clock(c, solverMethod)`	Solver clock	Operator 16.5

Operator 16.1 Clock

`Clock()`

Inferred clock. The operator returns a clock that is inferred.

[*Example:*

```
when Clock() then // equations are on the same clock
  x = A * previous(x) + B * u;
  Modelica.Utilities.Streams.print
    ("clock ticks at = " + String(sample(time)));
end when;
```

Note, in most cases, the operator is not needed and equations could be written without a **when***-clause (but not in the example above, since the* `print` *statement is otherwise not associated to a clock). This style is useful if a modeler would clearly like to mark the equations that must belong to one clock (although a tool could figure this out as well, if the* **when***-clause is not present).*]

Operator 16.2 Clock

`Clock(intervalCounter=`*intervalCounter*`, resolution=`*resolution*`)`

Rational interval clock. The first input argument, *intervalCounter*, is a clocked component expression (definition 16.6) or an evaluable expression of type `Integer` with `min = 0`. The optional second argument *resolution* (defaults to 1) is an evaluable expression of type `Integer` with `min = 1` and `unit = "Hz"`. If *intervalCounter* is an evaluable expression with value zero, the period of the clock is derived by clock inference, see section 16.7.5.

If *intervalCounter* is an evaluable expression greater than zero, the clock defines a periodic clock. If *intervalCounter* is a clocked component expression it must be greater than zero. The result is of base type `Clock` that ticks when `time` becomes t_{start}, $t_{\text{start}} + interval_1$, $t_{\text{start}} + interval_1 + interval_2$, ... The clock starts at the start of the simulation t_{start} or when the controller is switched on. At the start of the simulation, `previous(`*intervalCounter*`)` = *intervalCounter*`.start` and the clocks ticks the first time. At the first clock tick *intervalCounter* must be computed and the second clock tick is then triggered at $interval_1$ = *intervalCounter*/*resolution*. At the second clock tick at time $t_{\text{start}} + interval_1$, a new value for *intervalCounter* must be computed and the next clock tick is scheduled at $interval_2$ = *intervalCounter*/*resolution*, and so on.

[*The given interval and time shift can be modified by using the* `subSample`, `superSample`, `shiftSample` *and* `backSample` *operators on the returned clock, see section 16.5.2.*]

[*Example:*

```
  // first clock tick: previous(nextInterval) = 2
  Integer nextInterval(start = 2);
  Real y1(start = 0);
  Real y2(start = 0);
equation
  when Clock(2, 1000) then
    // periodic clock that ticks at 0, 0.002, 0.004, ...
    y1 = previous(y1) + 1;
  end when;

  when Clock(nextInterval, 1000) then
    // interval clock that ticks at 0, 0.003, 0.007, 0.012, ...
    nextInterval = previous(nextInterval) + 1;
    y2 = previous(y2) + 1;
  end when;
```

]

Note that operator `interval(c)` of `Clock c = Clock(nextInterval,` *resolution*`)` returns: `previous(`*intervalCounter*`) /` *resolution* (in seconds)

Operator 16.3 Clock

`Clock(interval=`*interval*`)`

Real interval clock. The input argument, *interval*, is a clocked component expression (definition 16.6) or a parameter expression. The *interval* must be strictly positive (*interval* > 0) of type `Real` with `unit = "s"`. The result is of base type `Clock` that ticks when `time` becomes t_{start}, $t_{\text{start}} + interval_1$, $t_{start} + interval_1 + interval_2$, ... The clock starts at the start of the simulation t_{start} or when the controller is switched on. Here the next clock tick is scheduled at $interval_1$ = `previous(`*interval*`)` = *interval*`.start`. At the second clock tick at time $t_{\text{start}} + interval_1$, the next clock tick is scheduled at $interval_2$ = `previous(`*interval*`)`, and so on. If *interval* is a parameter expression, the clock defines a periodic clock.

[*Note, the clock is defined with* `previous(`*interval*`)`*. Therefore, for sorting the input argument is treated as known. The given interval and time shift can be modified by using the* `subSample`, `superSample`, `shiftSample` *and* `backSample` *operators on the returned clock, see section 16.5.2. There are restrictions where this operator can be used, see* `Clock` *expressions below. Note that interval does not have to an evaluable expression, since different real interval clocks are never compared.*]

Operator 16.4 Clock

`Clock(condition=`*condition*`, startInterval=`*startInterval*`)`

Event clock. The first input argument, *condition*, is a continuous-time expression of type `Boolean`. The optional *startInterval* argument (defaults to 0) is the value returned by `interval()` at the first tick of the clock, see section 16.9. The result is of base type `Clock` that ticks when `edge(`*condition*`)` becomes `true`.

[*This clock is used to trigger a clocked base-partition due to a state event (that is a zero-crossing of a* `Real` *variable) in a continuous-time base-partition, or due to a hardware interrupt that is modeled as* `Boolean` *in the simulation model.*]

[*Example:*

```
Clock c = Clock(angle > 0, 0.1); // before first tick of c:
                                 // interval(c) = 0.1
```

]

[*The implicitly given interval and time shift can be modified by using the* `subSample`, `superSample`, `shiftSample` *and* `backSample` *operators on the returned clock, see section 16.5.2, provided the base interval is not smaller than the implicitly given interval.*]

Operator 16.5 `Clock`

`Clock(c=`*c*`, solverMethod=`*solverMethod*`)`

Solver clock. The first input argument, *c*, is a clock and the operator returns this clock. The returned clock is associated with the second input argument *solverMethod* of type `String`. The meaning of *solverMethod* is defined in section 16.8.2. If *solverMethod* is the empty `String`, then this `Clock` construct does not associate an integrator with the returned clock.

[*Example:*

```
Clock c1 = Clock(1, 10);                    // 100 ms, no solver
Clock c2 = Clock(c1, "ImplicitTrapezoid"); // 100 ms, ImplicitTrapezoid
    solver
Clock c3 = Clock(c2, "");                   // 100 ms, no solver
```

]

Besides inferred clocks and solver clocks, one of the following mutually exclusive associations of clocks are possible in one base-partition:

1. One or more periodic rational interval clocks, provided they are consistent with each other, see section 16.7.5.

   [*Example: Assume* `y = subSample(u)`*, and* `Clock(1, 10)` *is associated with* `u` *and* `Clock(2, 10)` *is associated with* `y`*, then this is correct, but it would be an error if* `y` *is associated with a* `Clock(1, 3)`.]

2. Exactly one non-periodic rational interval clock.

3. Exactly one real interval clock.

   [*Example: Assume* `Clock c = Clock(2.5)`*, then variables in the same base-partition can be associated multiple times with* `c` *but not multiple times with* `Clock(2.5)`.]

4. Exactly one event clock.

5. A default clock, if neither a real interval, nor a rational interval nor an event clock is associated with a base-partition. In this case the default clock is associated with the fastest sub-partition.

   [*Typically, a tool will use* `Clock(1.0)` *as a default clock and will raise a warning, that it selected a default clock.*]

Clock variables can be used in a restricted form of expressions. Generally, every expression switching between clock variables must be an evaluable expression (in order that clock analysis can be performed when translating a model). Thus subscripts on clock variables and conditions of if-then-else switching between clock variables must be evaluable expressions, and there are similar restrictions for sub-clock conversion operators section 16.5.2. Otherwise, the following expressions are allowed:

- Declaring arrays of clocks.

  [*Example:* `Clock c1[3] = {Clock(1), Clock(2), Clock(3)}`]

- Array constructors of clocks: `{}`, `[]`, `cat`.

- Array access of clocks.

  [*Example:* `sample(u, c1[2])`]

- Equality of clocks.

  [*Example:* `c1 = c2`]

- **`if`**-expressions of clocks in equations.

  [*Example:*

```
Clock c2 =
  if f > 0 then
    subSample(c1, f)
  elseif f < 0 then
    superSample(c1, f)
  else
    c1;
```

  ]

- Clock variables can be declared in models, blocks, connectors, and records. A clock variable can be declared with the prefixes `input`, `output`, **`inner`**, **`outer`**, but *not* with the prefixes `flow`, `stream`, `discrete`, `parameter`, or `constant`.

  [*Example:*

```
connector ClockInput = input Clock;
```

  ]

16.4 Clocked State Variables

Definition 16.7. ***Clocked state variable.*** A component expression which is not a parameter, and to which `previous` has been applied. □

The previous value of a clocked variable can be accessed with the `previous` operator, listed below.

Expression	*Description*	*Details*
`previous(`u`)`	Previous value of clocked variable	Operator 16.6

Operator 16.6 `previous`

`previous(`u`)`

The input argument u is a component expression (definition 16.6). If u is a parameter, its value is returned.

Otherwise: Input and return arguments are on the same clock. At the first tick of the clock of u or after a reset transition (see section 17.3.2), the start value of u is returned, see section 16.9. At subsequent activations of the clock of u, the value of u from the previous clock activation is returned.

[*At a clock tick only the (previous) values of the clocked state variables are needed to compute the new values of all clocked variables on that clock. This roughly corresponds to state variables in continuous time.*]

16.5 Partitioning Operators

A set of *clock conversion operators* together act as boundaries between different clock partitions.

16.5.1 Base-Clock Conversion Operators

The operators listed below convert between a continuous-time and a clocked-time representation and vice versa.

Expression	*Description*	*Details*
`sample(`*u*`, `*clock*`)`	Sample continuous-time expression	Operator 16.7
`hold(`*u*`)`	Zeroth order hold of clocked-time variable	Operator 16.8

Operator 16.7 `sample`

`sample(`*u*`, `*clock*`)`

Input argument *u* is a continuous-time expression according to section 3.8.6. The optional input argument *clock* is of type `Clock`, and can in a call be given as a named argument (with the name *clock*), or as positional argument. The operator returns a clocked variable that has *clock* as associated clock and has the value of the left limit of *u* when *clock* is active (that is the value of *u* just before the event of *clock* is triggered). If *clock* is not provided, it is inferred, see section 16.7.5.

[*Since the operator returns the left limit of u, it introduces an infinitesimal small delay between the continuous-time and the clocked base-partition. This corresponds to the reality, where a sampled data system cannot act infinitely fast and even for a very idealized simulation, an infinitesimal small delay is present. The consequences for the sorting are discussed below.*

Input argument u can be a general expression, because the argument is continuous-time and therefore has always a value. It can also be a constant, a parameter or a piecewise constant expression.

Note that `sample` *is an overloaded function: If* `sample` *has two positional input arguments and the second argument is of type* `Real`*, it is the operator from section 3.7.5. If* `sample` *has one input argument, or it has two input arguments and the second argument is of type* `Clock`*, it is the base-clock conversion operator from this section.*]

Operator 16.8 `hold`

`hold(`*u*`)`

Input argument *u* is a clocked (definition 16.3) component expression (definition 16.6) or a parameter expression. The operator returns a piecewise constant signal of the same type as *u*. When the clock of *u* ticks, the operator returns *u* and otherwise returns the value of *u* from the last clock activation. Before the first clock activation of *u*, the operator returns the start value of *u*, see section 16.9.

[*Since the input argument is not defined before the first tick of the clock of u, the restriction is present, that it must be a component expression (or a parameter expression), in order that the initial value of u can be used in such a case.*]

[*Example: Assume there is the following model:*

```
  Real y(start = 1), yc;
equation
  der(y) + y = 2;
  yc = sample(y, Clock(0.1));
initial equation
  der(y) = 0;
```

The value of `yc` *at the first clock tick is* $\texttt{yc} = 2$ *(and not* $\texttt{yc} = 1$*). The reason is that the continuous-time model* `der(y) + y = 2` *is first initialized and after initialization* `y` *has the value 2. At the first clock tick at* $\texttt{time} = 0$*, the left limit of* `y` *is 2 and therefore* $\texttt{yc} = 2$*.*]

16.5.1.1 Sorting of a Simulation Model

[*Since* `sample(u)` *returns the left limit of* `u`*, and the left limit of* `u` *is a known value, all inputs to a base-partition are treated as known during sorting. Since a periodic and interval clock can tick at most once at a time instant, and since the left limit of a variable does not change during event iteration (i.e., re-evaluating a base-partition associated with a condition clock always gives the same result because the* `sample(u)` *inputs do not change and therefore need not to be re-evaluated), all base-partitions, see section 16.7.3, need not to be sorted with respect to each other. Instead, at an event instant, active base-partitions can be evaluated first (and once) in any order. Afterwards, the continuous-time base-partition is evaluated.*

Event iteration takes place only over the continuous-time base-partition. In such a scenario, accessing the left limit of `u` *in* `sample(u)` *just means to pick the latest available value of* `u` *when the base-partition is entered, storing it in a local variable of the base-partition and only using this local copy during evaluation of the equations in this base-partition.*]

16.5.2 Sub-Clock Conversion Operators

The operators listed below convert between synchronous clocks.

Expression	*Description*	*Details*
`subSample(`u`, factor)`	Clock that is slower by a factor	Operator 16.9
`superSample(`u`, factor)`	Clock that is faster by a factor	Operator 16.10
`shiftSample(`u`, shiftCounter, resolution)`	Clock with time-shifted ticks	Operator 16.11
`backSample(`u`, backCounter, resolution)`	Inverse of `shiftSample`	Operator 16.12
`noClock(`u`)`	Clock that is always inferred	Operator 16.13

These operators have the following properties:

- The input argument u is a clocked expression or an expression of type `Clock`. (The operators can operate on all types of clocks.) If u is a clocked expression, the operator returns a clocked variable that has the same type as the expression. If u is an expression of type `Clock`, the operator returns a `Clock` – except for `noClock` where it is an error.
- The optional input arguments `factor` (defaults to 0, with `min = 0`), and `resolution` (defaults to 1, with `min = 1`) are evaluable expressions of type `Integer`.
- Calls of the operators can use named arguments for the multi-letter arguments (i.e., not for u) with the given names, or positional arguments.

  [*Named arguments can make the calls easier to understand.*]
- The input arguments `shiftCounter` and `backCounter` are evaluable expressions of type `Integer` with `min = 0`.

Operator 16.9 subSample

`subSample(`u`, factor=`*factor*`)`

The clock of `y = subSample(`u`,` *factor*`)` is *factor* times slower than the clock of u. At every *factor* ticks of the clock of u, the operator returns the value of u. The first activation of the clock of `y` coincides with the first activation of the clock of u, and then every activation of the clock of `y` coincides with the every *factor*th activativation of the clock of u. If *factor* is not provided or is equal to zero, it is inferred, see section 16.7.5.

Operator 16.10 superSample

`superSample(`u`, factor=`*factor*`)`

The clock of `y = superSample(`u`,` *factor*`)` is *factor* times faster than the clock of u. At every tick of the clock of `y`, the operator returns the value of u from the last tick of the clock of u. The first activation of the clock of `y` coincides with the first activation of the clock of u, and then the interval between activations of the clock of u is split equidistantly into *factor* activations, such that the activation $1 + k \cdot factor$ of `y` coincides with the $1 + k$ activation of u.

[*Thus* `subSample(superSample(`u`,` *factor*`),` *factor*`)` $= u$.]

If *factor* is not provided or is equal to zero, it is inferred, see section 16.7.5. If an event clock is associated to a base-partition, all its sub-partitions must have resulting clocks that are sub-sampled with an `Integer` factor with respect to this base-clock.

[*Example:*

```
Clock u = Clock(x > 0);
Clock y1 = subSample(u, 4);
Clock y2 = superSample(y1, 2); // fine; y2 = subSample(u, 2)
Clock y3 = superSample(u, 2);  // error
Clock y4 = superSample(y1, 5); // error
```

]

Operator 16.11 shiftSample

`shiftSample(`*u*`, shiftCounter=`*k*`, resolution=`*resolution*`)`

The operator `c = shiftSample(`*u*`, `*k*`, `*resolution*`)` splits the interval between ticks of *u* into *resolution* equidistant intervals *i*. The clock `c` then ticks *k* intervals *i* after each tick of *u*.

It leads to

```
shiftSample(u, k, resolution) =
  subSample(shiftSample(superSample(u, resolution), k), resolution)
```

[*Note, due to the restriction of* `superSample` *on event clocks,* `shiftSample` *can only shift the number of ticks of the event clock, but cannot introduce new ticks. Example:*

```
// Rational interval clock
Clock u  = Clock(3, 10);             // ticks: 0, 3/10, 6/10, ...
Clock y1 = shiftSample(u, 1, 3);     // ticks: 1/10, 4/10, ...
// Event clock
Integer revolutions = integer(time);
Clock u = Clock(change(revolutions), startInterval = 0.0);
                                     // ticks: 0.0, 1.0, 2.0, 3.0, ...
Clock y1 = shiftSample(u, 2);        // ticks: 2.0, 3.0, ...
Clock y2 = shiftSample(u, 2, 3);     // error (resolution must be 1)
```

Additional example showing the full form:

```
  Integer intervalCnt(start=2);
  Integer cnt(start=0);
  Clock u = Clock(intervalCnt,1);
  Clock s1 = shiftSample(u, 3, 2);
equation
   when u then
     cnt = previous(cnt) + 1;
     intervalCnt = if (cnt>=2) then 1 else previous(intervalCnt);
   end when;
```

Here `u` *ticks at 0, 2, 3, 4, 5, 6. First you* `superSample` *to split each sampling interval in two equal parts leading to the ticks 0.0, 1.0, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0. Then the simple* `shiftSample` *removes the first three ticks giving 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0. And finally every other point is removed by* `subSample`*, and* `s1` *ticks at 2.5, 3.5, 4.5, 5.5.*]

Operator 16.12 backSample

`backSample(`*u*`, backCounter=`*cnt*`, resolution=`*res*`)`

The input argument *u* is either a component expression (definition 16.6) or an expression of type `Clock`. This is an inverse of `shiftSample` such that `Clock y = backSample(`*u*`, `*cnt*`, `*res*`)` implicitly defines a clock `y` such that `shiftSample(y, `*cnt*`, `*res*`)` activates at the same times as *u*. It is an error if the clock of `y` starts before the base-clock of *u*.

At every tick of the clock of `y`, the operator returns the value of *u* from the last tick of the clock of *u*. If *u* is a clocked component expression, the operator returns the start value of *u*, see section 16.9, before the first tick of the clock of *u*.

[*Example:*

```
// Rational interval clock 1
Clock u  = Clock(3, 10);           // ticks: 0, 3/10, 6/10, ...
Clock y1 = shiftSample(u, 3);      // ticks: 9/10, 12/10, ...
Clock y2 = backSample(y1, 2);      // ticks: 3/10, 6/10, ...
Clock y3 = backSample(y1, 4);      // error (ticks before u)
Clock y4 = shiftSample(u, 2, 3);   // ticks: 2/10, 5/10, ...
Clock y5 = backSample(y4, 1, 3);   // ticks: 1/10, 4/10, ...
// Event clock
Integer revolutions = integer(time);
```

```
Clock u = Clock(change(revolutions), startInterval = xx)
                                    // ticks: 0, 1.0, 2.0, 3.0, ...
Clock y1 = shiftSample(u, 3);       // ticks: 3.0, 4.0, ...
Clock y2 = backSample(y1, 2);       // ticks: 1.0, 2.0, ...
```

]

Operator 16.13 noClock

`noClock(`*u*`)`

The clock of `y = noClock(`*u*`)` is always inferred, and *u* must be part of the same base-clock as `y`. At every tick of the clock of `y`, the operator returns the value of *u* from the last tick of the clock of *u*. If `noClock(`*u*`)` is called before the first tick of the clock of *u*, the start value of *u* is returned.

[*Clarification of* `backSample`*:*

Let a and b be positive integers with $a < b$*, and*

```
yb = backSample(u, a, b)
ys = shiftSample(u, b − a, b)
```

Then when `ys` *exists, also* `yb` *exists and* `ys = yb`.

The variable `yb` *exists for the above parameterization with* `a < b` *one clock tick before* `ys`. *Therefore,* `backSample` *is basically a* `shiftSample` *with a different parameterization and the clock of* `backSample.y` *ticks before the clock of* `u`. *Before the clock of* `u` *ticks,* `yb = u.start`.]

[*Clarification of* `noClock` *operator:*

Note, that `noClock(u)` *is not equivalent to* `sample(hold(u))`. *Consider the following model:*

```
model NoClockVsSampleHold
  Clock clk1 = Clock(0.1);
  Clock clk2 = subSample(clk1, 2);
  Real x(start = 0), y(start = 0), z(start = 0);
equation
  when clk1 then
    x = previous(x) + 0.1;
  end when;
  when clk2 then
    y = noClock(x);       // most recent value of x
    z = sample(hold(x)); // left limit of x (infinitesimally delayed)!
  end when;
end NoClockVsSampleHold;
```

Due to the infinitesimal delay of `sample`, `z` *will not show the current value of* `x` *as* `clk2` *ticks, but will show its previous value (left limit). However,* `y` *will show the current value, since it has no infinitesimal delay.*]

Note that it is not legal to compute the derivative of the `sample`, `subSample`, `superSample`, `backSample`, `shiftSample`, and `noClock` operators.

16.6 Clocked When-Clause

In addition to the previously discussed **when**-equation (see section 8.3.5), a *clocked* **when**-clause is introduced:

```
when clockExpression then
  ⟨clocked equations⟩
  ...
end when;
```

The clocked **when**-clause cannot be nested and does not have any **elsewhen** part. It cannot be used inside an algorithm. General equations are allowed in a clocked **when**-clause.

For a clocked **when**-clause, all equations inside the **when**-clause are clocked with the same clock given by the *clockExpression*.

16.7 Clock Partitioning

This section defines how clock-partitions and clocks associated with equations are inferred.

[*Typically clock partitioning is performed before sorting the equations. The benefit is that clocking and symbolic transformation errors are separated.*]

Every clocked variable is uniquely associated with exactly one clock.

After model flattening, every equation in an equation section, every expression and every algorithm section is either continuous-time, or it is uniquely associated with exactly one clock. In the latter case it is called a *clocked equation*, a *clocked expression* or *clocked algorithm* section respectively. The associated clock is either explicitly defined by a **when**-clause, see section 16.5.2, or it is implicitly defined by the requirement that a clocked equation, a clocked expression and a clocked algorithm section must have the same clock as the variables used in them with exception of the expressions used as first arguments in the conversion operators of section 16.5. *Clock inference* means to infer the clock of a variable, an equation, an expression or an algorithm section if the clock is not explicitly defined and is deduced from the required properties in the previous two paragraphs.

All variables in an expression without clock conversion operators must have the same clock to infer the clocks for each variable and expression. The clock inference works both forward and backwards regarding the data flow and is also being able to handle algebraic loops. The clock inference method uses the set of variable incidences of the equations, i.e., what variables that appear in each equation.

Note that incidences of the first argument of clock conversion operators of section 16.5 are handled specially.

[*As clock partitions are solely determined by the equations, two different clock partitions can have clocks defined by the same expressions. It is a quality of implementation issue that such partitions are executed synchronously, e.g., by putting them in the same task in a real-time simulation context.*]

16.7.1 Flattening of Model

The clock partitioning is conceptually performed after model flattening, i.e., redeclarations have been elaborated, arrays of model components expanded into scalar model components, and overloading resolved. Furthermore, function calls to inline functions have been inlined.

[*This is called* conceptually, *because a tool might do this more efficiently in a different way, provided the result is the same as if everything is flattened. For example, array and matrix equations and records don't not need to be expanded if they have the same clock.*]

Furthermore, each non-trivial expression (non-literal, non-constant, non-parameter, non-variable), $expr_i$, appearing as first argument of a clock conversion operator (except `hold` and `backSample`) is recursively replaced by a unique variable, v_i, and the equation $v_i = expr_i$ is added to the equation set.

16.7.2 Connected Components of the Equations and Variables Graph

Consider the set E of equations and the set V of unknown variables (not constants and parameters) in a flattened model, i.e., $M = \langle E, V \rangle$. The partitioning is described in terms of an undirected graph $\langle N, F \rangle$ with the nodes N being the set of equations and variables, $N = E \cup V$. The set incidence(e) for an equation e in E is a subset of V, in general, the unknowns which lexically appear in e. There is an edge in F of the graph between an equation, e, and a variable, v, if $v \in$ incidence(e):

$$F = \{(e, v) : e \in E, v \in \text{incidence}(e)\}$$

A set of clock partitions is the *connected components* (Wikipedia, *Connected components*) of this graph with appropriate definition of the incidence operator.

A special case is the built-in variable `time` (see section 3.6.7). Each use of `time` is conceptually included as a separate variable in this analysis, $time_i$ with `der`($time_i$) = 1.

[*This means that* `time` *can be used in different partitions without any restrictions. Additionally, it means that every sub-partition directly referencing* `time` *contains a call to* `der`.]

16.7.3 Base-Partitioning

The goal is to identify all clocked equations and variables that should be executed together in the same task, as well as to identify the continuous-time base-partition.

The base-partitioning is performed with base-clock inference which uses the following incidence definition:
incidence(e) =

> the *unknown* variables, as well as variables `x` in `der(x)`, `pre(x)`, and `previous(x)`, which lexically appear in e
>
> > except as first argument of base-clock conversion operators: `sample` and `hold` and `Clock(condition=..., startInterval=...)`.

The resulting set of connected components, is the partitioning of the equations and variables, $B_i = \langle E_i, V_i \rangle$, according to base-clocks and continuous-time partitions.

The base-partitions are identified as *clocked* or as *continuous-time partitions* according to the following properties:

A variable `u` in `sample(u)`, a variable `y` in `y = hold(ud)`, and a variable `b` in `Clock(b, startInterval=...)` where the `Boolean` `b` is in a continuous-time partition.

Correspondingly, variables `u` and `y` in `y = sample(uc)`, `y = subSample(u)`, `y = superSample(u)`, `y = shiftSample(u)`, `y = backSample(u)`, `y = previous(u)`, are in a clocked base-partition. Equations in a clocked **`when`**-clause are also in a clocked base-partition. Other base-partitions where none of the variables in the partition are associated with any of the operators above have an unspecified partition kind and are considered continuous-time base-partitions.

All continuous-time base-partitions are collected together and form *the continuous-time base-partition.*

[*Example:*

```
// Controller 1
ud1 = sample(y,c1);
0 = f1(yd1, ud1, previous(yd1));

// Controller 2
ud2 = superSample(yd1,2);
0 = f2(yd2, ud2);

// Continuous-time system
u = hold(yd2);
0 = f3(der(x1), x1, u);
0 = f4(der(x2), x2, x1);
0 = f5(der(x3), x3);
0 = f6(y, x1, u);
```

After base-partitioning, the following partitions are identified:

```
// Base partition 1 -- clocked partition
ud1 = sample(y, c1);                // incidence(e) = {ud1}
0 = f1(yd1, ud1, previous(ud1)); // incidence(e) = {yd1, ud1}
ud2 = superSample(yd1, 2);          // incidence(e) = {ud2, yd1}
0 = f2(yd2, ud2);                   // incidence(e) = {yd2, ud2}

// Base partition 2 -- continuous-time partition
u = hold(yd2);                      // incidence(e) = {u}
0 = f3(der(x1), x1, u);             // incidence(e) = {x1, u}
0 = f4(der(x2), x2, x1);            // incidence(e) = {x2, x1}
0 = f6(y, x1, u);                   // incidence(e) = {y, x1, u}

// Identified as separate partition, but belonging to base-partition 2
0 = f5(der(x3), x3);                // incidence(e) = {x3}
```

]

16.7.4 Sub-Partitioning

For each clocked base-partition B_i, identified in section 16.7.3, the sub-partitioning is performed with sub-clock inference which uses the following incidence definition:

incidence(e) =
the *unknown* variables, as well as variables `x` in `der(x)`, `pre(x)`, and `previous(x)`, which lexically appear in e
except as first argument of sub-clock conversion operators: `subSample`, `superSample`, `shiftSample`, `backSample`, `noClock`, and `Clock` with first argument of `Boolean` type.

The resulting set of connected components, is the partitioning of the equations and variables, $S_{ij} = \langle E_{ij}, V_{ij} \rangle$, according to sub-clocks.

The connected components (corresponding to the sub-clocks) are then further split into strongly connected components corresponding to systems of equations. The resulting sets of equations and variables shall be possible to solve separately, meaning that systems of equations cannot involve different sub-clocks.

It can be noted that:

$$E_{ij} \bigcap E_{kl} = \emptyset, \forall i \neq k, j \neq l$$
$$V_{ij} \bigcap V_{kl} = \emptyset, \forall i \neq k, j \neq l$$
$$V = \bigcup V_{ij}$$
$$E = \bigcup E_{ij}$$

[*Example: After sub-partitioning of the example from section 16.7.3, the following partitions are identified:*

```
// Base partition 1 (clocked partition)
// Sub-partition 1.1
ud1 = sample(y, c1);              // incidence(e) = {ud1}
0 = f1(yd1, ud1, previous(yd1)); // incidence(e) = {yd1,ud1}

// Sub-partition 1.2
ud2 = superSample(yd1, 2);        // incidence(e) = {ud2}
0 = f2(yd2, ud2);                 // incidence(e) = {yd2,ud2}

// Base partition 2 (no sub-partitioning, since continuous-time)
u = hold(yd2);
0 = f3(der(x1), x1, u);
0 = f4(der(x2), x2, x1);
0 = f5(der(x3), x3);
0 = f6(y, x1, u);
```

]

[*Example: Forbidding systems of equations involving different sub-clocks means that the following is forbidden:*

```
  Real a;
  //Real x=a+z;
  Real y=superSample(a+z, 2);
  Real z;
equation
  a+z = sample(time, Clock(1,100));
  0 = subSample(y, 2)+a;
```

Here `a` *and* `z` *are part of one sub-clock, and* `y` *of another, and the system of equations involve both of them.*

The following legal example solves the issues in the previous example by replacing `a` *by* `x-z` *(and simplifying the equations). Additionally, it shows that it is not required that the sub-clocks can necessarily be sorted:*

```
Real x=sample(time, Clock(1,100));
Real y=superSample(x, 2);
Real z=subSample(y, 2)+x;
```

Here `x` *and* `z` *are part of one sub-partition, and* `y` *of another. The equations form three equation systems with one equation in each (hence trivially satisfying the requirement that only variables from one sub-partition are being solved). The equation systems need to be solved in a strict order, but the first and last equation system belong to one sub-clock, while the second equation system belongs to another sub-clock. This illustrates that there is no guarantee that the sub-partitions can be ordered in agreement with the equation systems. Note that equation systems with more than one equation are also allowed in sub-partitions.*]

16.7.5 Sub-Clock Inferencing

For each base-partition, the base interval needs to be determined and for each sub-partition, the sub-sampling factors and shift need to be determined. The sub-partition intervals are constrained by `subSample` and `superSample` factors which might be known (or evaluable expression) or unspecified, as well as by `shiftSample`, `shiftCounter` and `resolution`, or `backSample`, `backCounter` and `resolution`. This constraint set is used to solve for all intervals and sub-sampling factors and shift of the sub-partitions. The model is erroneous if no solution exist.

[*It must be possible to determine that the constraint set is valid at compile time. However, in certain cases, it could be possible to defer providing actual numbers until run-time.*]

It is required that accumulated sub- and supersampling factors in the range of 1 to 2^{63} can be handled.

[*64 bit internal representation of numerator and denominator with sign can be used and gives minimum resolution* 1.08×10^{-19} *seconds and maximum range* 9.22×10^{18} *seconds* $= 2.92 \times 10^{11}$ *years.*]

16.8 Discretized Sub-Partition

[*The goal is that every continuous-time Modelica model can be utilized in a sampled data control system. This is achieved by solving the continuous-time equations with a defined integration method between clock ticks. With this feature, it is for example possible to invert the nonlinear dynamic model of a plant, see Thümmel, Looye, Kurze, Otter, and Bals (2005), and use it in a feedforward path of an advanced control system that is associated with a clock.*

This feature also allows defining multi-rate systems: Different parts of the continuous-time model are associated to different clocks and are solved with different integration methods between clock ticks, e.g., a very fast sub-system with an implicit solver with a small step-size and a slow sub-system with an explicit solver with a large step-size.]

With the language elements defined in this section, continuous-time equations can be used in clocked partitions. Hereby, the continuous-time equations are solved with the defined integration method between clock ticks.

Such a sub-partition is called a *discretized* sub-partition, and the clock ticks are not interpreted as events, but as step-sizes of the integrator that the integrator must hit exactly. Hence, no event handling is triggered at clock ticks (provided an explicit event is not triggered from the model at this time instant).

[*The interpretation of the clock ticks is the same assumption as for manually discretized controllers, such as the z-transform.*]

[*It is not defined how to handle events that are triggered while solving a discretized sub-partition. For example, a tool could handle events in the same way as for a usual simulation – but only check them at the time associated with clock-ticks.*

Alternatively, relations might be interpreted literally, so that events are no longer triggered (in order that the time for an integration step is always the same, as needed for hard real-time requirements). However,

even if relations do not generate events, when-clauses and operators `edge` *and* `change` *should behave as normal.*]

From the viewpoint of other partitions, the discretized continuous-time variables only have values at clock ticks (internally it may be more complicated, see section 16.8.2). Therefore, outside the discretized sub-partitions themselves, they are treated similarly to discrete-time sub-partitions. Especially, operators such as `sample`, `hold`, `subSample` must be used to communicate signals of the discretized sub-partition with other partitions.

16.8.1 Discrete-time and Discretized Sub-Partitions

Additionally to the variability of expressions defined in section 3.8, an orthogonal concept *clocked variability* is defined in this section. If not explicitly stated otherwise, an expression with a variability such as *continuous-time* or *discrete-time* means that the expression is inside a partition that is not associated to a clock. If an expression is present in a base-partition that is not a continuous-time base-partition, it is a *clocked expression* and has *clocked variability*.

After sub-clock inferencing, see section 16.7.5, every sub-partition that is associated with a clock has to be categorized as *discrete-time* or *discretized*.

[*Previously, discrete-time sub-partition was refered to as* clocked discrete-time partition*, and discretized sub-partition as* clocked discretized continuous-time partition.]

If a sub-partition that is not continuous-time contains any of the operators `der`, `delay`, `spatialDistribution`, or event related operators from section 3.7.5 (with exception of `noEvent` and `smooth`), or contains a `when`-clause with a `Boolean` condition, it is a *discretized* sub-partition. Otherwise, it is a *discrete-time* sub-partition.

[*That is, a discrete-time sub-partition is a standard sampled data system described by difference equations.*]

A discretized sub-partition has to be solved with a *solver method* of section 16.8.2. When `previous(x)` is used on a continuous-time state variable `x`, then `previous(x)` uses the start value of `x` as value for the first clock tick.

The use of the operator `sample` from section 3.7.5 in a discretized sub-partition is problematic. A diagnostic is recommended, especially if the operator is intended to generate events faster than the clock ticks, and otherwise the sampling should ideally be adjusted to the clock ticks.

[*The reason for not disallowing* `sample` *in a discretized sub-partition is to make it possible to include any continuous-time Modelica model in a sampled data control system. Note that even if the sampling is slower than the clock ticks (or even the same rate) it still introduces the problem of possibly uneven sampling.*]

In a discrete-time sub-partition none of the event generating mechanisms apply. Especially neither relations, nor any of the built-in operators of section 3.7.2 (event triggering mathematical functions) will trigger events.

16.8.2 Solver Methods

A sub-partition can have an integration method, directly associated (section 16.8.3) or inferred from other sub-partitions (section 16.8.4). A predefined type `ModelicaServices.Types.SolverMethod` defines the methods supported by the respective tool by using the `choices` annotation.

[*The* `ModelicaServices` *package contains tool specific definitions. A string is used instead of an enumeration, since different tools might have different values and then the integer mapping of an enumeration is misleading since the same value might characterize different integrators.*]

The following names of solver methods are standardized:

```
type SolverMethod = String annotation(choices(
  choice="External" "Solver specified externally",
  choice="ExplicitEuler" "Explicit Euler method (order 1)",
  choice="ExplicitMidPoint2" "Explicit mid point rule (order 2)",
  choice="ExplicitRungeKutta4" "Explicit Runge-Kutta method (order 4)",
```

```
  choice="ImplicitEuler" "Implicit Euler method (order 1)",
  choice="ImplicitTrapezoid" "Implicit trapezoid rule (order 2)"
)) "Type of integration method to solve differential equations in a " +
   "discretized sub-partition."
```

If a tool supports one of the integrators of `SolverMethod`, it must use the solver method name of above.

[*A tool may also support other integrators. Typically, a tool supports at least methods* `"External"` *and* `"ExplicitEuler"`. *If a tool does not support the integration method defined in a model, typically a warning message is printed and the method is changed to* `"External"`.]

If the solver method is `"External"`, then the sub-partition associated with this method is integrated by the simulation environment for an interval of length of `interval()` using a solution method defined in the simulation environment.

[*An example of such a solution method could be to have a table of the clocks that are associated with discretized sub-partitions and a method selection per clock. In such a case, the solution method might be a variable step solver with step-size control that integrates between two clock ticks. The simulation environment might also combine all partitions associated with method* `"External"`*, as well as all continuous-time partitions, and integrate them together with the solver selected by the simulation environment.*]

If the solver method is *not* `"External"`, then the sub-partition is integrated using the given method with the step-size `interval()`.

[*For a periodic clock, the integration is thus performed with fixed step size.*]

The solvers are defined with respect to the underlying ordinary differential equation in state space form that corresponds to the sub-partition before it has been discretized, at least conceptually:

$$\dot{x} = f(x, u, t)$$
$$y = g(x, u, t)$$

where:

- t is time
- $u_c(t)$ is the continuous-time `Real` vector of input variables
- $u_d(t)$ is the discrete-time `Real`/`Integer`/`Boolean`/`String` vector of input variables
- $x(t)$ is the continuous-time real vector of states
- $y(t)$ is the continuous-time or discrete-time `Real`/`Integer`/`Boolean`/`String` vector of algebraic and/or output variables

A solver method is applied to a discretized sub-partition. Such a partition has explicit inputs u marked by `sample(`u`)`, `subSample(`u`)`, `superSample(`u`)`, `shiftSample(`u`)` and/or `backSample(`u`)`. Furthermore, the outputs y of such a partition are marked by `hold(`y`)`, `subSample(`y`)`, `superSample(`y`)`, `shiftSample(`y`)`, and/or `backSample(`y`)`. The arguments of these operators are to be used as input signals u and output signals y in the conceptual ordinary differential equation above, and in the discretization formulae below, respectively.

The solver methods (with exception of `"External"`) are defined by integrating from clock tick t_{i-1} to clock tick t_i and computing the desired variables at t_i, with $h = t_i - t_{i-1} =$ `interval(`u`)` and $x_i = x(t_i)$ (for all methods: $y_i = g(x_i, u_{c,i}, u_{d,i}, t_i)$):

SolverMethod	*Solution method*
"ExplicitEuler"	$x_i := x_{i-1} + h \cdot \dot{x}_{i-1}$ $\dot{x}_i := f(x_i, u_{c,i}, u_{d,i}, t_i)$
"ExplicitMidPoint2"	$x_i := x_{i-1} + h \cdot f(x_{i-1} + \frac{1}{2} \cdot h \cdot \dot{x}_{i-1}, \frac{u_{c,i-1} + u_{c,i}}{2}, u_{d,i-1}, t_{i-1} + \frac{1}{2} \cdot h)$ $\dot{x}_i := f(x_i, u_{c,i}, u_{d,i}, t_i)$
"ExplicitRungeKutta4"	$k_1 := h \cdot \dot{x}_{i-1}$ $k_2 := h \cdot f(x_{i-1} + \frac{1}{2}k_1, \frac{u_{c,i-1} + u_{c,i}}{2}, u_{d,i-1}, t_{i-1} + \frac{1}{2} \cdot h)$ $k_3 := h \cdot f(x_{i-1} + \frac{1}{2}k_2, \frac{u_{c,i-1} + u_{c,i}}{2}, u_{d,i-1}, t_{i-1} + \frac{1}{2} \cdot h)$ $k_4 := h \cdot f(x_{i-1} + k_3, u_{c,i}, u_{d,i}, t_i)$ $x_i := x_{i-1} + \frac{1}{6} \cdot (k_1 + 2 \cdot k_2 + 2 \cdot k_3 + k_4)$ $\dot{x}_i := f(x_i, u_{c,i}, u_{d,i}, t_i)$
"ImplicitEuler"	Equation system with unknowns: x_i, $\dot{x}_i$ $x_i = x_{i-1} + h \cdot \dot{x}_i$ $\dot{x}_i = f(x_i, u_{c,i}, u_{d,i}, t_i)$
"ImplicitTrapezoid"	Equation system with unknowns: x_i, $\dot{x}_i$ $x_i = x_{i-1} + \frac{1}{2}h \cdot (\dot{x}_i + \dot{x}_{i-1})$ $\dot{x}_i = f(x_i, u_{c,i}, u_{d,i}, t_i)$

The initial conditions will be used at the first tick of the clock, and the first integration step will go from the first to the second tick of the clock.

[*Example: Assume the differential equation*

```
  input Real u;
  Real x(start = 1, fixed = true);
equation
  der(x) = -x + u;
```

shall be transformed to a discretized sub-partition with the "ExplicitEuler" *method. The following model is a manual implementation:*

```
  input Real u;
  parameter Real x_start = 1;
  Real x(start = x_start); // previous(x) = x_start at first clock tick
  Real der_x(start = 0);   // previous(der_x) = 0 at first clock tick
protected
  Boolean first(start = true);
equation
  when Clock() then
    first = false;
    if previous(first) then
      // first clock tick (initialize system)
      x = previous (x);
    else
      // second and further clock tick
      x = previous(x) + interval() * previous(der_x);
    end if;
    der_x = -x + u;
  end when;
```

]

[*For the implicit integration methods the efficiency can be enhanced by utilizing the discretization formula during the symbolic transformation of the equations. For example, linear differential equations are then mapped to linear and not non-linear algebraic equation systems, and also the structure of the equations can be utilized. For details see Elmqvist, Otter, and Cellier (1995). It might be necessary to associate additional data for an implicit integration method, e.g., the relative tolerance to solve the non-linear algebraic equation systems, or the maximum number of iterations in case of hard realtime requirements.*

This data is tool specific and is typically either defined with a vendor annotation or is given in the simulation environment.]

16.8.3 Associating a Solver to a Sub-Partition

A `SolverMethod` can be associated to a clock with the overloaded `Clock` constructor `Clock(`*c*`, solverMethod =...)`, see section 16.3. If a clock is associated with a sub-clock of a discretized sub-partition and a `SolverMethod` is associated with this clock, then the sub-partition is integrated with it.

[*Example:*

```
// Continuous PI controller in a clocked partition
vd = sample(x2, Clock(Clock(1, 10), solverMethod="ImplicitEuler"));
e = ref - vd;
der(xd) = e / Ti;
u = k * (e + xd);

// Physical model
f = hold(u);
der(x1) = x2;
m * der(x2) = f;
```

]

16.8.4 Inferencing of solverMethod

If a `solverMethod` is not explicitly associated with a sub-partition, it is inferred with a similar mechanism as for sub-clock inferencing, see section 16.7.5.

First, one set is constructed for each sub-partition, containing just this sub-partition. These sets are then merged as follows: For each set without a specified `solverMethod`, the set is merged with sets connected to it (these may contain a `solverMethod`), and this is repeated until it is not possible to merge more sets. The sets connected in this way should be part of the same base-partition and connected through a sub-clock conversion operator (`subSample`, `superSample`, `shiftSample`, `backSample`, or `noClock`).

- It is an error if this set contains multiple different values for `solverMethod`.
- If the set contains continuous-time equations:
  - It is an error if this set contains no `solverMethod`.
  - Otherwise, the specified `solverMethod` is used.
- If the set does not contain continuous-time equations, there is no need for a `solverMethod`. However, inferencing between sub-partitions works the same regardless of whether there are continuous-time equations.

[*Example:*

```
model InferenceTest
  Real x(start = 3) "Explicitly using ExplicitEuler";
  Real y "Explicitly using ImplicitEuler method";
  Real z "Inferred to use ExplicitEuler";
equation
  der(x) = -x + sample(1, Clock(Clock(1, 10), solverMethod="ExplicitEuler"));
  der(y) = subSample(x, 2) +
           sample(1, Clock(Clock(2, 10), solverMethod="ImplicitEuler"));
  der(z) = subSample(x, 2) + 1;
end InferenceTest;
```

Note that there is only one base-partition, but it has two different periodic rational clocks - consistent with section 16.7.5.

```
model IllegalInference
  Real x(start = 3) "Explicitly using ExplicitEuler";
  Real y "Explicitly using ImplicitEuler method";
```

```
  Real z;
equation
  der(x) = -x + sample(1, Clock(Clock(1, 10), solverMethod="ExplicitEuler"));
  der(y) = subSample(x, 2) +
           sample(1, Clock(Clock(2, 10), solverMethod="ImplicitEuler"));
  der(z) = subSample(x, 4) + 1 + subSample(y);
end IllegalInference;
```

Here `z` *is a continuous-time equation connected directly to both* `x` *and* `y` *sub-partitions that have different* `solverMethod`.]

16.9 Initialization of Clocked Partitions

The standard scheme for initialization of Modelica models does not apply for clocked base-partitions. Instead, initialization is performed in the following way:

- Variables in clocked partitions cannot be used in initial equation or initial algorithm sections.
- Attribute `fixed` cannot be applied to variables in clocked partitions. The attribute `fixed` is true for variables to which `previous` is applied, otherwise false.

16.10 Other Operators

A few additional utility operators are listed below.

Expression	*Description*	*Details*
`firstTick`(u)	Test for first clock tick	Operator 16.14
`interval`(u)	Interval between previous and present tick	Operator 16.15

It is an error if these operators are called in the continuous-time base-partition.

Operator 16.14 `firstTick`

`firstTick`(u)

This operator returns true at the first tick of the clock of the expression, in which this operator is called. The operator returns false at all subsequent ticks of the clock. The optional argument u is only used for clock inference, see section 16.7.

Operator 16.15 `interval`

`interval`(u)

This operator returns the interval between the previous and present tick of the clock of the expression, in which this operator is called. The optional argument u is only used for clock inference, see section 16.7. At the first tick of the clock the following is returned:

1. If the specified clock interval is a parameter expression, this value is returned.
2. Otherwise the start value of the variable specifying the interval is returned.
3. For an event clock the additional `startInterval` argument to the event clock constructor is returned.

The return value of `interval` is a scalar `Real` number.

[*Example: A discrete PI controller is parameterized with the parameters of a continuous PI controller, in order that the discrete block is robust against changes in the sample period. This is achieved by discretizing a continuous PI controller (here with an implicit Euler method):*

```
block ClockedPI
  parameter Real T "Time constant of continuous PI controller";
  parameter Real k "Gain of continuous PI controller";
  input Real u;
  output Real y;
  Real x(start = 0);
```

```
  protected
  Real Ts = interval(u);
equation
  /* Continuous PI equations: der(x) = u / T; y = k * (x + u);
   * Discretization equation: der(x) = (x - previous (x)) / Ts;
   */
  when Clock() then
    x = previous (x) + Ts / T * u;
    y = k * (x + u);
  end when;
end ClockedPI;
```

A continuous-time model is inverted, discretized and used as feedforward controller for a PI controller (`der`*,* `previous`*,* `interval` *are used in the same partition):*

```
block MixedController
  parameter Real T "Time constant of continuous PI controller";
  parameter Real k "Gain of continuous PI controller";
  input Real y_ref, y_meas;
  Real y;
  output Real yc;
  Real z(start = 0);
  Real xc(start = 1, fixed = true);
  Clock c = Clock(Clock(0.1), solverMethod="ImplicitEuler");
protected
  Real uc;
  Real Ts = interval(uc);
equation
  /* Continuous-time, inverse model */
  uc = sample(y_ref, c);
  der(xc) = uc;
  /* PI controller */
  z = if  firstTick() then 0 else
  previous(z) + Ts / T * (uc - y_meas);
  y = xc + k * (xc + uc);
  yc = hold (y);
end MixedController;
```

]

16.11 Semantics

The execution of sub-partitions requires exact time management for proper synchronization. The implication is that testing a `Real`-valued time variable to determine sampling instants is not possible. One possible method is to use counters to handle sub-sampling scheduling,

```
Clock_i_j_ticks =
  if pre(Clock_i_j_ticks) < subSamplingFactor_i_j then
    1 + pre(Clock_i_j_ticks)
  else
    1;
```

and to test the counter to determine when the sub-clock is ticking:

```
Clock_i_j_activated =
  BaseClock_i_activated and Clock_i_j_ticks >= subSamplingFactor_i_j;
```

The `Clock_`i`_`j`_activated` flag is used as the guard for the sub partition equations.

[*Consider the following example:*

```
model ClockTicks
  Integer second = sample(1, Clock(1));
  Integer seconds(start = -1) = mod(previous(seconds) + second, 60);
```

```
  Integer milliSeconds(start = -1) =
    mod(previous(milliSeconds) + superSample(second, 1000), 1000);
  Integer minutes(start = -1) =
    mod(previous(minutes) + subSample(second, 60), 60);
end ClockTicks;
```

A possible implementation model is shown below using Modelica 3.2 semantics. The base-clock is determined to 0.001 seconds and the sub-sampling factors to 1000 and 60000.

```
model ClockTicksWithModelica32
   Integer second;
   Integer seconds(start = -1);
   Integer milliSeconds(start = -1);
   Integer minutes(start = -1);

   Boolean BaseClock_1_activated;
   Integer Clock_1_1_ticks(start = 59999);
   Integer Clock_1_2_ticks(start = 0);
   Integer Clock_1_3_ticks(start = 999);
   Boolean Clock_1_1_activated;
   Boolean Clock_1_2_activated;
   Boolean Clock_1_3_activated;
equation
  // Prepare clock tick
  BaseClock_1_activated =  sample(0, 0.001);
  when BaseClock_1_activated then
    Clock_1_1_ticks =
      if pre(Clock_1_1_ticks) < 60000 then 1 + pre(Clock_1_1_ticks) else 1;
    Clock_1_2_ticks =
      if pre(Clock_1_2_ticks) < 1 then 1 + pre(Clock_1_2_ticks) else 1;
    Clock_1_3_ticks =
      if pre(Clock_1_3_ticks) < 1000 then 1 + pre(Clock_1_3_ticks) else 1;
  end when;
  Clock_1_1_activated =  BaseClock_1_activated and Clock_1_1_ticks >= 60000;
  Clock_1_2_activated =  BaseClock_1_activated and Clock_1_2_ticks >= 1;
  Clock_1_3_activated =  BaseClock_1_activated and Clock_1_3_ticks >= 1000;

  // ------------------------------------------------------------------------
  // Sub partition execution
  when {Clock_1_3_activated} then
    second = 1;
  end when;
  when {Clock_1_1_activated} then
    minutes = mod(pre(minutes) + second, 60);
  end when;
  when {Clock_1_2_activated} then
    milliSeconds = mod(pre(milliSeconds) + second, 1000);
  end when;
  when {Clock_1_3_activated} then
    seconds = mod(pre(seconds) + second, 60);
  end when;
end ClockTicksWithModelica32;
```

]

Chapter 17

State Machines

This chapter defines language elements to define clocked state machines. These state machines have a similar modeling power as Statecharts (Harel 1987) and have the important feature that at one clock tick, there is only one assignment to every variable (for example, it is an error if state machines are executed in parallel and they assign to the same variable at the same clock tick; such errors are detected during translation). Furthermore, it is possible to activate and deactivate clocked equations and blocks at a clock tick. An efficient implementation will only evaluate the equations and blocks that are active at the current clock tick. With other Modelica language elements, this important feature cannot be defined.

The semantics of the state machines defined in this chapter is inspired by mode automata and is basically the one from Lucid Synchrone 3.0 (Pouzet 2006). Note, safety critical control software in aircrafts is often defined with such kind of state machines. The following properties are different to Lucid Synchrone 3.0:

- Lucid Synchrone has two kinds of transitions: *strong* and *weak* transitions. Strong transitions are executed before the actions of a state are evaluated and weak transitions are executed after the actions of a state are evaluated. This can lead to surprising behavior, because the actions of a state are skipped if it is activated by a weak transition and exited by a true strong transition.

  For this reason, the state machines in this chapter use *immediate* (= the same as *strong*) and *delayed* transitions. Delayed transitions are *immediate* transitions where the condition is automatically delayed with an implicit `previous`.

- Parallel state machines can be explicitly synchronized with a language element (similarly as parallel branches in Sequential Function Charts). This often occurring operation can also be defined in Statecharts or in Lucid Synchrone state machines but only indirectly with appropriate conditions on transitions.

- Modelica blocks can be used as states. They might contain clocked equations. If the equations are discretized, they are integrated between the previous and the current clock tick, if the corresponding state is active.

17.1 Transitions

Any Modelica block instance without continuous-time equations or continuous-time algorithms can potentially be a state of a state machine. A cluster of instances which are coupled by `transition` statements makes a state machine. All parts of a state machine must have the same clock. All transitions leaving one state must have different priorities. One and only one instance in each state machine must be marked as initial by appearing in an `initialState` statement.

The special kinds of `connect`-like equations listed below are used to define define a state machine.

Expression	*Description*	*Details*
`transition`(*from*, *to*, *condition*, ...)	State machine transition between states	Operator 17.1
`initialState`(*state*)	State machine initial state	Operator 17.2

The `transition`- and `initialState`-equations can only be used in equations, and cannot be used inside **`if`**-equations with conditions that are not parameter expressions, or in **`when`**-equations.

The operators listed below are used to query the status of the state machine.

Expression	*Description*	*Details*
`activeState(`*state*`)`	Predicate for active state	Operator 17.3
`ticksInState()`	Ticks since activation	Operator 17.4
`timeInState()`	Time since activation	Operator 17.5

Operator 17.1 `transition`

```
transition(from, to, condition,
  immediate=imm, reset=reset, synchronize=synch, priority=prio)
```

Arguments *from* and *to* are block instances, and *condition* is a `Boolean` argument. The optional arguments `immediate`, `reset`, and `synchronize` are of type `Boolean`, have parameter variability and a default of `true`, `true`, `false` respectively. The optional argument `priority` is of type `Integer`, has parameter variability and a default of 1.

This operator defines a transition from instance *from* to instance *to*. The *from* and *to* instances become states of a state machine. The transition fires when *condition* = `true` if *imm* = `true` (this is called an *immediate transition*) or `previous(`*condition*`)` when *imm* = `false` (this is called a *delayed transition*). Argument `priority` defines the priority of firing when several transitions could fire. In this case the transition with the smallest value of `priority` fires. It is required that *prio* ≥ 1 and that for all transitions from the same state, the priorities are different. If *reset* = `true`, the states of the target state are reinitialized, i.e., state machines are restarted in initial state and state variables are reset to their start values. If *synch* = `true`, any transition is disabled until all state machines of the from-state have reached final states, i.e., states without outgoing transitions. For the precise details about firing a transition, see section 17.3.

Operator 17.2 `initialState`

```
initialState(state)
```

Argument *state* is the block instance that is defined to be the initial state of a state machine. At the first clock tick of the state machine, this state becomes active.

Operator 17.3 `activeState`

```
activeState(state)
```

Argument *state* is a block instance. The operator returns `true` if this instance is a state of a state machine and this state is active at the actual clock tick. If it is not active, the operator returns `false`.

It is an error if the instance is not a state of a state machine.

Operator 17.4 `ticksInState`

```
ticksInState()
```

Returns the number of ticks of the clock of the state machine for which the currently active state has maintained its active state without interruption, i.e., without local or hierarchical transitions from this state. In the case of a self-transition to the currently active state or to an active enclosing state, the number is reset to one.

This function can only be used in state machines.

Operator 17.5 `timeInState`

```
timeInState()
```

Returns the time duration as `Real` in [s] for which the currently active state has maintained its active state without interruption, i.e., without local or hierarchical transitions from this state. In the case of a self-transition to the currently active state or to an active enclosing state, the time is reset to zero.

This function can only be used in state machines.

[*Example: If there is a transition with* `immediate = false` *from state* `A1` *to* `A2` *and the condition is* `ticksInState() >= 5`*, and* `A1` *became active at 10ms, and the clock period is 1ms, then* `A1` *will be active at 10ms, 11ms, 12ms, 13ms, 14ms, and will be not active at 15 ms.*

```
  block State end State;
  State A0;
  State A1; // Becomes active at 10ms
  State A2;
equation
  initialState(A0);
  transition(A0, A1, sample(time, Clock(1, 1000)) > 0.0095);
  transition(A1, A2, ticksInState() >= 5, immediate = false);
```

]

17.2 State Machine Graphics

[*Figure 17.1 shows the recommended layout of a state machine.*]

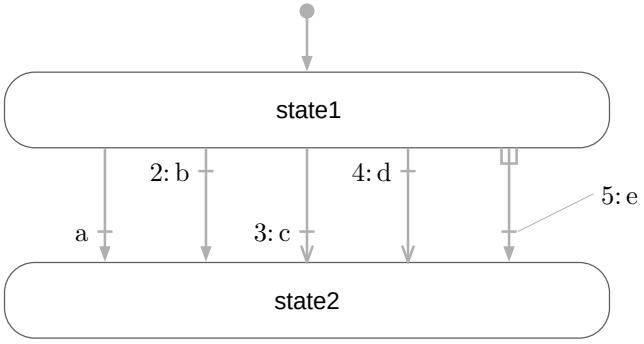


Figure 17.1: *Recommended layout of a simple state machine. For the 5 transitions, the settings are as follows, from left to right:* `immediate` *= true, false, true, false, true;* `reset` *= true, true, false, false, true;* `synchronize` *= false, false, false, false, true;* `priority` *= 1, 2, 3, 4, 5.*

The annotation for graphics of `transition` has the following structure: `annotation(Line(...), Text(...))`; and for `initialState()`: *graphical-primitives*`(Line(...))`; with `Line` and `Text` annotations defined in chapter 18.

[*Example:*

```
transition(state2, state1, x < 10,
           immediate = true, reset = true, synchronize = false, priority = 1)
  annotation(
    Line(
      points = {{-40,-16},{-36,-4},{-32,8},{-40,26},{-40,32},{-46,50}},
      color = {175, 175, 175},
      thickness = 0.25,
      smooth = Smooth.Bezier),
    Text(
      string = "%condition",
      extent = {{4, -4}, {4, -10}},
      fontSize = 10,
      textStyle = {TextStyle.Bold},
      textColor = {95, 95, 95},
      horizontalAlignment = TextAlignment.Left),
  );
```

]

The `Text` annotation representing the transition condition can use the notation `%condition` to refer to the condition expression.

The extent of the Text is interpreted relative to either the first point of the `Line`, in the case of `immediate = false`, or the last point (`immediate = true`).

In addition to the line defined by the points of the `Line` annotation, a perpendicular line is used to represent the transition. This line is closer to the first point if `immediate = false` otherwise closer to the last point.

If the condition text is somewhat distant from the perpendicular line, a dimmed straight line joins the transition text and the perpendicular line. (See the rightmost transition above.)

If `reset = true`, a filled arrow head is used otherwise an open arrow head. For `synchronize = true`, an inverse "fork" symbol is used in the beginning of the arrow. (See the rightmost transition above.)

The value of the `priority` attribute is prefixing the condition text followed by a colon if `priority` > 1.

The `initialState` line has a filled arrow head and a bullet at the opposite end of the initial state (as shown above).

17.3 State Machine Semantics

For the purpose of defining the semantics of state machines, assume that the data of all transitions are stored in an array of records:

```
record Transition
  Integer from;
  Integer to;
  Boolean immediate = true;
  Boolean reset = true;
  Boolean synchronize = false;
  Integer priority = 1;
end Transition;
```

The transitions are sorted with lowest priority number last in the array; and the priorities must be unique for each value of `from`. The states are enumerated from 1 and up. The transition conditions are stored in a separate array `c[:]` since they are time varying.

The semantics model is a discrete-time system with inputs {`c[:]`, `active`, `reset`} with `t` being an array corresponding to the inputs to the transition operator, outputs {`activeState`, `activeReset`, `activeResetStates[:]`} and states {`nextState`, `nextReset`, `nextResetStates[:]`}. For a top-level state machine, active is always true. For sub-state machines, active is true only when the parent state is active. For a top-level state machine, reset is true at the first activation only. For sub-state machine, reset is propagated from the state machines higher up.

17.3.1 State Activation

The state update starts from `nextState`, i.e., what has been determined to be the next state at the previous time. `selectedState` takes into account if a reset of the state machine is to be done.

```
output Integer selectedState = if reset then 1 else previous(nextState);
```

The integer fired is calculated as the index of the transition to be fired by checking that `selectedState` is the from-state and the condition is true for an immediate transition or `previous(`*condition*`)` is true for a delayed transition. The max function returns the index of the transition with highest priority or 0.

```
Integer fired =
  max(
    if t[i].from == selectedState and
        (if t[i].immediate then c[i] else previous(c[i]))
      then i
      else 0
    for i in 1 : size(t, 1));
```

The start value of c is false. This definition would require that the previous value is recorded for all transitions conditions. Below is described an equivalent semantics which just require to record the value of one integer variable delayed.

The integer immediate is calculated as the index of the immediate transition to potentially be fired by checking that `selectedState` is the from-state and the condition is true. The max function returns the index of the transition with true condition and highest priority or 0.

```
Integer immediate =
  max(
    if t[i].immediate and t[i].from == selectedState and c[i] then i else 0
    for i in 1 : size(t, 1));
```

In a similar way, the `Integer delayed` is calculated as the index for a potentially delayed transition, i.e., a transition taking place at the next clock tick. In this case the from-state needs to be equal to `nextState`:

```
Integer delayed =
  max(
    if not t[i].immediate and t[i].from == nextState and c[i] then i else 0
    for i in 1 : size(t, 1));
```

The transition to be fired is determined as follows, taking into account that a delayed transition might have higher priority than an immediate:

```
Integer fired = max(previous(delayed), immediate);
```

`nextState` is set to the found transitions to-state:

```
Integer nextState =
  if active then
    (if fired > 0 then t[fired].to else selectedState)
  else
    previous(nextState);
```

In order to define synchronize transitions, each state machine must determine which are the final states, i.e., states without from-transitions and to determine if the state machine is in a final state currently:

```
Boolean finalStates[nStates] =
  {min(t[j].from <> i for j in 1 : size(t, 1)) for i in 1 : nStates};
Boolean stateMachineInFinalState = finalStates[activeState];
```

To enable a synchronize transition, all the `stateMachineInFinalState` conditions of all state machines within the meta state must be true. An example is given below in the semantic example model.

17.3.2 Reset Handling

A state can be reset for two reasons:

- The whole state machine has been reset from its context. In this case, all states must be reset, and the initial state becomes active.
- A reset transition has been fired. Then, its target state is reset, but not other states.

The first reset mechanism is handled by the `activeResetStates` and `nextResetStates` vectors.

The state machine reset flag is propagated and maintained to each state individually:

```
output Boolean activeResetStates[nStates] =
  {reset or previous(nextResetStates[i]) for i in 1 : nStates};
```

until a state is eventually executed, then its corresponding reset condition is set to false:

```
Boolean nextResetStates[nStates] =
  if active then
    {activeState <> i and activeResetStates[i] for i in 1 : nStates}
  else
    previous(nextResetStates)
```

The second reset mechanism is implemented with the **selectedReset** and **nextReset** variables. If no reset transition is fired, the **nextReset** is set to false for the next cycle.

17.3.3 Activation Handling

When a state is suspended, its equations should not be executed, and its variables keep their values – including state-variables in discretized equations.

The execution of a sub-state machine has to be suspended when its enclosing state is not active. This activation flag is given as a **Boolean** input **active**. When this flag is true, the sub-state machine maintains its previous state, by guarding the equations of the state variables **nextState**, **nextReset** and **nextResetStates**.

17.3.4 Semantics Summary

The entire semantics model is given below:

```
model StateMachineSemantics "Semantics of state machines"
  parameter Integer nStates;
  parameter Transition t[:] "Array of transition data sorted in priority";
  input Boolean c[size(t, 1)] "Transition conditions sorted in priority";
  input Boolean active "true if the state machine is active";
  input Boolean reset "true when the state machine should be reset";
  Integer selectedState = if reset then 1 else previous(nextState);
  Boolean selectedReset = reset or previous(nextReset);
  // For strong (immediate) and weak (delayed) transitions
  Integer immediate =
    max(
      if (t[i].immediate and t[i].from == selectedState and c[i]) then i else 0
      for i in 1 : size(t, 1));
  Integer delayed =
    max(
      if (not t[i].immediate and t[i].from == nextState and c[i]) then i else 0
      for i in 1 : size(t, 1));
  Integer fired = max(previous(delayed), immediate);
  output Integer activeState =
    if reset then 1 elseif fired > 0 then t[fired].to else selectedState;
  output Boolean activeReset =
    reset or (if fired > 0 then t[fired].reset else selectedReset);

  // Update states
  Integer nextState = if active then activeState else previous(nextState);
  Boolean nextReset = not active and previous(nextReset);
  // Delayed resetting of individual states
  output Boolean activeResetStates[nStates] =
    {reset or previous(nextResetStates[i]) for i in 1 : nStates};
  Boolean nextResetStates[nStates] =
    if active then
      {activeState <> i and activeResetStates[i] for i in 1 : nStates}
    else
      previous(nextResetStates);
  Boolean finalStates[nStates] =
    {min(t[j].from <> i for j in 1 : size(t, 1)) for i in 1 : nStates};
  Boolean stateMachineInFinalState = finalStates[activeState];
end StateMachineSemantics;
```

17.3.5 Merging Variable Definitions

[*When a state class uses an* **outer** output *declaration, the equations have access to the corresponding variable declared* **inner**. *Special rules are then needed to maintain the single assignment rule since multiple definitions of such outer variables in different mutually exclusive states needs to be merged.*]

In each state, the outer output variables are solved for and for each such variable a single definition is formed:

```
v :=
  if activeState(state₁) then
    expre₁
  elseif activeState(state₂) then
    expre₂
  elseif ...
  else
    last(v)
```

`last` is special internal semantic operator returning its input. It is just used to mark for the sorting that the incidence of its argument should be ignored. A start value must be given to the variable if not assigned in the initial state.

A new assignment equation is formed which might be merged on higher levels in nested state machines.

17.3.6 Merging Connections to Outputs

[*The causal connection semantics of Modelica for non-state machines are generalized to states of state machines, using the fact that only one state is active at a time.*]

It is possible to connect outputs each coming from different states of state machines together – and connect this with other causal connectors. These outputs are combined seen as one source of the signal, and give the following constraint equations,

```
u₁ = u₂ = ... = y₁ = y₂ = ...
```

with y_i being outputs from different states of the state-machine and u_i being other causal variables. The semantics is defined similarly to section 17.3.5:

```
v = if activeState(state₁) then
      y₁
    elseif activeState(state₂) then
      y₂
    elseif ...
    else
      last(v);
u₁ = v
u₂ = v
...
```

17.3.7 Example

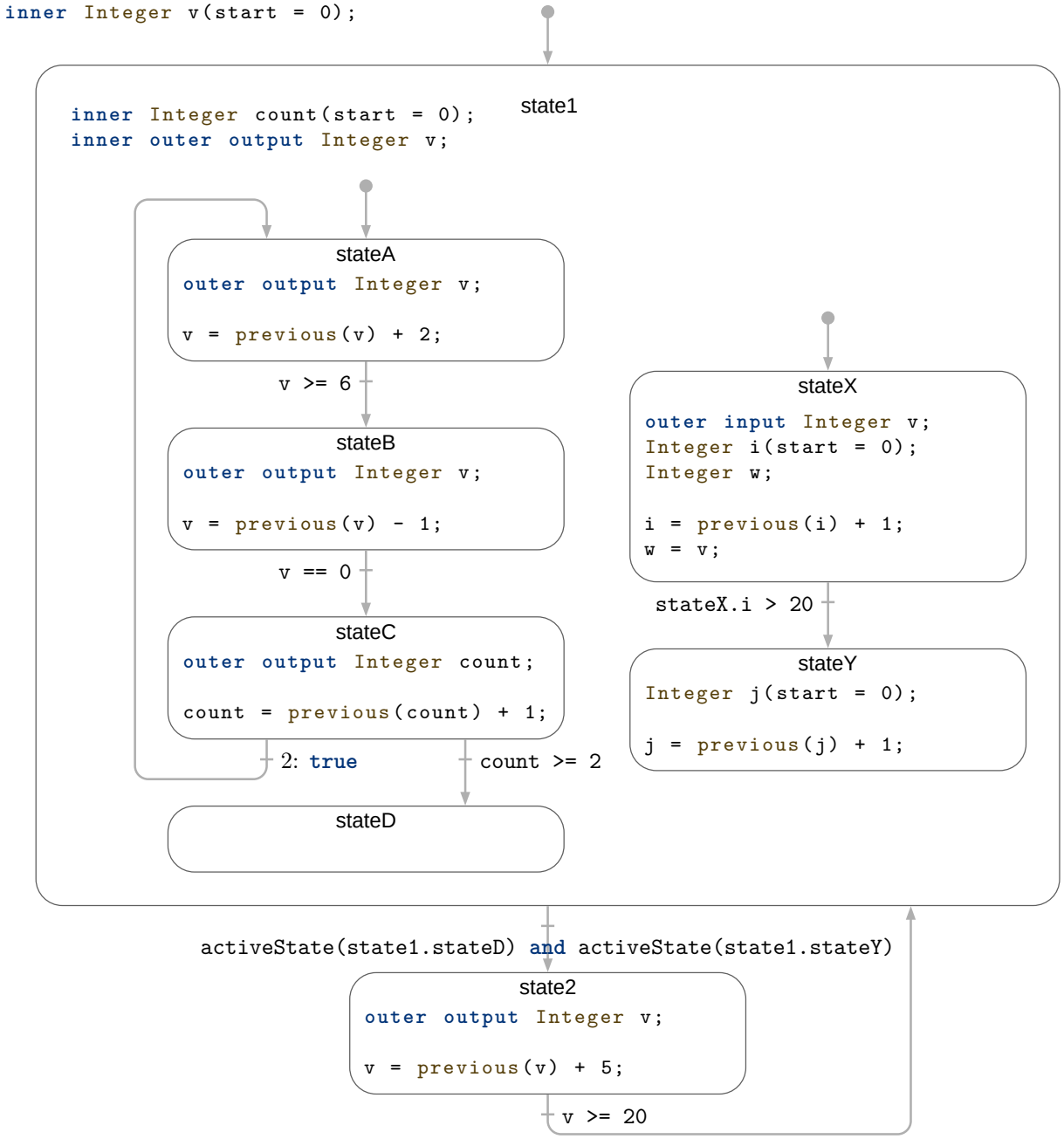


Figure 17.2: *Example of a hierarchical state machine.*

[*Example: Consider the hierarchical state machine in figure 17.2. The model demonstrates the following properties:*

- `state1` *is a meta state with two parallel state machines in it.*
- `stateA` *declares* `v` *as* **`outer`** `output`. `state1` *is on an intermediate level and declares* `v` *as* **`inner`** **`outer`** `output`, *i.e., matches lower level* **`outer`** `v` *by being* **`inner`** *and also matches higher level* **`inner`** `v` *by being* **`outer`**. *The top level declares* `v` *as* **`inner`** *and gives the start value.*
- `count` *is defined with a start value in* `state1`. *It is reset when a reset transition (*`v >= 20`*) is made to* `state1`.
- `stateX` *declares the local variable* `w` *to be equal to* `v` *declared as* **`inner`** `input`.

- `stateY` *declares a local counter* `j`*. It is reset at start and as a consequence of the reset transition (*`v >= 20`*) to* `state1`*: When the reset transition (*`v >= 20`*) fires, then the variables of the active states are reset immediately (so* `count` *from* `state1`*, and* `i` *from* `stateX`*). The variables of other states are only reset at the time instants when these states become active. So* `j` *in* `StateY` *is reset to 0, when the transition* `stateX.i > 20` *fires (after* `state1` *became active again, so after the reset transition* `v >= 20`*).*
- *Synchronizing the exit from the two parallel state machines of* `state1` *is done by checking that* `stated` *and* `stateY` *are active using the* `activeState` *function.*

The Modelica code (without annotations) is:

```
block HierarchicalAndParallelStateMachine
  inner Integer v(start = 0);

  State1 state1;
  State2 state2;
equation
  initialState(state1);
  transition(state1, state2,
             activeState(state1.stateD) and activeState(state1.stateY),
             immediate = false);
  transition(state2, state1, v >= 20, immediate = false);

public
  block State1
    inner Integer count(start = 0);
    inner outer output Integer v;

    block StateA
      outer output Integer v;
    equation
      v = previous(v) + 2;
    end StateA;
    StateA stateA;

    block StateB
      outer output Integer v;
    equation
      v = previous(v) - 1;
    end StateB;
    StateB stateB;

    block StateC
      outer output Integer count;
    equation
      count = previous(count) + 1;
    end StateC;
    StateC stateC;

    block StateD
    end StateD;
    StateD stateD;

  equation
    initialState(stateA);
    transition(stateA, stateB, v >= 6, immediate = false);
    transition(stateB, stateC, v == 0, immediate = false);
    transition(stateC, stateA, true, immediate = false, priority = 2);
    transition(stateC, stateD, count >= 2, immediate = false);

  public
    block StateX
      outer input Integer v;
```

```
      Integer i(start = 0);
      Integer w; // = v;
    equation
      i = previous(i) + 1;
      w = v;
    end StateX;
    StateX stateX;

    block StateY
      Integer j(start = 0);
    equation
      j = previous(j) + 1;
    end StateY;
    StateY stateY;

  equation
    initialState(stateX);
    transition(stateX, stateY, stateX.i > 20,
               immediate = false, reset = false);
  end State1;

  block State2
    outer output Integer v;
  equation
    v = previous(v) + 5;
  end State2;
end HierarchicalAndParallelStateMachine;
```

Figure 17.3 shows the behavior of the state machine.

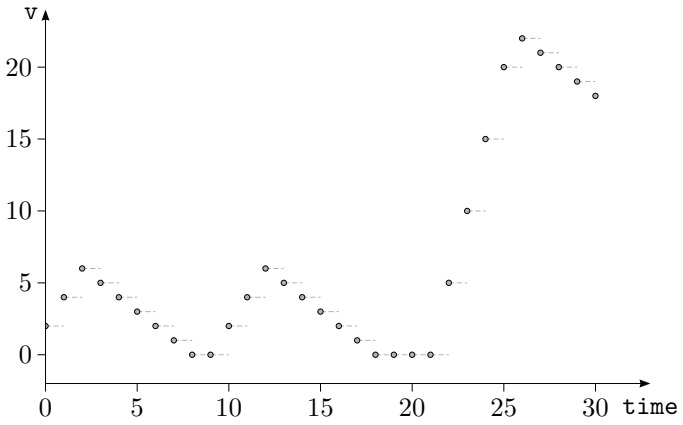


Figure 17.3: *State machine behavior, as reflected by the variable* `v`.

The transition from `state1` *to* `state2` *could have been done with a* `synchronize` *transition with* `condition =true` *instead. The semantically equivalent model is shown below:*

```
block HierarchicalAndParallelStateMachine
  extends StateMachineSemantics(
    nStates = 2,
    t = {Transition(from = 1, to = 2, immediate = false, synchronize = true),
         Transition(from = 2, to = 1, immediate = false)},
    c = {true, v >= 20});
  Boolean init(start = true) = sample(false);

  block State1
    Boolean active;
    Boolean reset;
    outer input Integer v_previous;
    inner output Integer v;
```

```
  inner Integer count(start = 0);
  inner Integer count_previous = if reset then 0 else previous(count);

  block StateMachineOf_stateA
    extends StateMachineSemantics(
      nStates = 4,
      t = {Transition(from = 1, to = 2, immediate = false),
           Transition(from = 2, to = 3, immediate = false),
           Transition(from = 3, to = 1, immediate = false),
           Transition(from = 3, to = 4, immediate = false)},
      c = {v >= 6, v == 0, true, count >= 2});
    outer input Integer v_previous;
    outer output Integer v;
    outer input Integer count_previous;
    outer output Integer count;
  equation
    inFinalState = true; // no macro states
    if activeState == 1 then
      // equations for stateA
      v = v_previous + 2;
      count = count_previous;
    elseif activeState == 2 then
      // equations for stateB
      v = v_previous - 1;
      count = count_previous;
    elseif activeState == 3 then
      // equations for stateC
      v = v_previous;
      count = count_previous + 1;
    else // if activeState == 4 then
      // equations for stateD
      v = v_previous;
      count = count_previous;
    end if;
  end StateMachineOf_stateA;

  StateMachineOf_stateA stateMachineOf_stateA(
    active = active, reset = reset);

  block StateMachineOf_stateX
    extends StateMachineSemantics(
      nStates = 2,
      t = {Transition(from = 1, to = 2, immediate = false, reset = false)},
      c = {i > 25});
    outer input Integer v;
    Integer i(start = 0);
    Integer i_previous;
    Integer j(start = 0);
    Integer j_previous;
    Integer w;
  equation
    inFinalState = true; // no macro states
    if activeState == 1 then
      // equations for stateX
      i_previous =
        if activeReset or activeResetStates[1] then 0 else previous(i);
      j_previous = previous(j);
      i = i_previous + 1;
      w = v;
      j = j_previous;
    else // if activeState == 2 then
      // equations for stateY
      i_previous = previous(i);
```

```
        j_previous =
          if activeReset or activeResetStates[2] then 0 else previous(j);
        i = i_previous;
        w = previous(w);
        j = j_previous + 1;
      end if;
    end StateMachineOf_stateX;

    StateMachineOf_stateX stateMachineOf_stateX(
      active = active, reset = reset);
    Boolean inFinalState =
      stateMachineOf_stateA.stateMachineInFinalState and
      stateMachineOf_stateX.stateMachineInFinalState;
  end State1;

  State1 state1;
  Integer v(start = 0);
  inner Integer v_previous = if reset then 0 else previous(v);
equation
  active = true;
  reset = previous(init);
  if activeState == 1 then
    // equations for state1
    inFinalState = state1.inFinalState;
    state1.active = true;
    state1.reset = activeReset or activeResetStates[1];
    v = state1.v;
  else // if activeState == 2 then
    // equations for state2
    inFinalState = true; // not macro state
    state1.active = false;
    state1.reset = false;
    v = previous(v) + 5;
  end if;
end HierarchcialAndParallelStateMachine;
```

]

Chapter 18

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 18.2.

Annotations are optional in the Modelica grammar, and when present, indicated using the **annotation** keyword, see *annotation-clause* in the grammar (appendix A.2.7). The structure of the annotation content is the same as a class modification (*class-modification* in the grammar). (For replaceable class declarations with a *constraining-clause* also refer to section 7.3.2.1.) The specification in this document defines the semantic meaning if a tool implements any of these annotations.

18.1 Notation for Annotation Definitions

Annotations are defined using the syntactic forms of Modelica component declarations and record definitions, with the following special semantics. If the annotation is described by a component declaration, the annotation is used in the form of a value modifier for the same name. If the annotation is described by a **record** *class* the annotation is used in the form of a modifier for a **record** *component* with the same name.

A declaration equation for a component or record member specifies a default to be used when no corresponding annotation is given. Default behavior can also be specified in the text as an alternative to a declaration equation. Component declaration annotations always have default behavior, as all annotations are optional. When there is no declaration equation and no other explanation of default behavior in the text for a record member, an annotation modifier for the record must contain a modifier for that member.

When an annotation is defined with a component variability prefix (section 4.4.3), this restricts the allowed variability of corresponding annotation modifiers analogously to the rules in section 3.8. If the annotation is declared as an `/* evaluable */ parameter` the corresponding modifier is further restricted to be evaluable. If the annotation is declared as a `/* literal */ constant` the corresponding modifier is further restricted to be a literal value.

18.2 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. The only requirement is that any tool shall save files with all vendor-specific annotations (and all annotations from this chapter) intact. 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.*]

18.3 Documentation

The `Documentation` annotation has the following contents, where the `info` and `revisions` annotations are described in section 18.3.1, and the `figures` annotation is described in section 18.3.2:

```
record Documentation
  /*literal*/ constant String info = "" "Description of the class";
  /*literal*/ constant String revisions = "" "Revision history";
  Figure[:] figures = {}; "Simulation result figures";
  String[:] styleSheets = {} "Style sheets for documentation";
end Documentation;
```

The `styleSheets` may also be given as a single string, see section 18.3.1.1.

How the tool interprets the information in `Documentation` is unspecified.

18.3.1 Class Description and Revision History

Inside the `Documentation` annotation, the `info` annotation gives a textual description of the class, and the `revisions` annotation gives a revision history.

[*The* `revisions` *documentation may be omitted in printed documentation.*]

If the string starts with the tag `<html>` or `<HTML>` the entire string is HTML encoded (and is assumed to end with `</html>` or `</HTML>` and shall be rendered as HTML even if the end-tags are missing), otherwise the entire string is rendered as is. The HTML encoded content may contain links. For external links, see section 13.5. Links to Modelica classes may be defined with the HTML link command using scheme `Modelica` (using its lower case form in the URI, see section 13.5), 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. User-defined fragment specifiers (anchors) may also be used, and they may be renamed when generating HTML (in particular to avoid collisions). Example:

```
<a href="modelica:/MultiBody.Joints.Revolute#info">Revolute</a>
```

18.3.1.1 Style Sheets

Inside the `Documentation` annotation, each element of the `styleSheets` annotation array specifies a cascading style sheet. The style sheets are used when displaying the `info` and `revisions` annotations found in the `Documentation` annotations of the package. The `styleSheets` annotation is only considered for top-level packages, and applies to the entire package. The style sheets will be cascaded in the given order. Specifying just a string for `styleSheets` has the same meaning as specifying a singleton array containing the string.

[*It is recommended to use* `class` *and* `id` *selectors with a NameOfLibrary- prefix to avoid collisions when the content is included in a larger context.*]

The style sheet rules should not use type or universal selectors, due to possible interference with tool-specific styling.

Vendors should use a *NameOfVendor-* prefix to style vendor generated HTML content surrounding the user provided documentation. If tools want to give users (of that tool) the possibility to override the tool-specific CSS they can document that. The prefix is used to avoid this happening by accident.

18.3.2 Figures

Inside the **Documentation** annotation, each element of the **figures** annotation array has the following content:

```
record Figure
  /*literal*/ constant String title = "" "Title meant for display";
  /*literal*/ constant String identifier = "" "Identifier meant for
    programmatic access";
  /*literal*/ constant String group = "" "Name of figure group";
  /*literal*/ constant Boolean preferred = false "Automatically display figure
    after simulation";
  Plot[:] plots "Plots";
  /*literal*/ constant String caption = "" "Figure caption";
end Figure;
```

A **Figure** is a graphical container that can contain several **plots** described by **Plot** annotations:

```
record Plot
  /*literal*/ constant String title "Title meant for display";
  /*literal*/ constant String identifier = "" "Identifier meant for
    programmatic access";
  Curve[:] curves "Plot curves";
  /*literal*/ constant Axis x "X axis properties";
  /*literal*/ constant Axis y "Y axis properties";
end Plot;
```

A **Plot** can contain several **curves**, see section 18.3.2.2, that all share a common **x** and **y** axis with properties described in section 18.3.2.1.

Both **Figure** and **Plot** can have an optional title. When the **Figure title** is the empty string (the default), the tool must produce a non-empty title based on the figure content. On the other hand, the **Plot title** has a tool-dependent default, but the default may be the empty string. When the **Plot title** is the empty string, no title should be shown. The plot title is not to be confused with the plot *label* which is never empty, see below. Variable replacements, as described in section 18.3.2.4, can be used in the **title** of **Figure** and **Plot**.

The **identifier** in **Figure** and **Plot** is a String identifier, and is intended to identify the **Figure** and **Plot** for programmatic access. The **figures** annotation is inherited in the sense that each class has a collection of figures comprised by the contents of the **figures** annotation in the class itself, as well as the **figures** annotations from any base classes. A **Figure** must be uniquely identified by its **identifier** and a class having it in its collection. This means that a **Figure identifier** must be unique among all **Figure** annotations within the same **figures** annotation as well as among all **figures** annotations from inherited classes. A **Plot identifier** on the other hand is only required to be unique among the **plots** in the the same **Figure** annotation. If an **identifier** is an empty string it cannot be used for programmatic access and is exempt from the uniqueness requirements.

[*For* **Figure**, *this makes it possible to reference the plot from a tool-specific scripting environment. For* **Plot**, *this makes it possible to reference the plot in the figure caption, which becomes useful when the* **Figure** *contains more than one* **Plot**.]

Even though a **Figure** annotation can be shared through inheritance between classes in a class hierarchy, note that each simulated class provides its own data to be displayed in the figure.

Every **Plot** has an automatically generated *label* which is required to be shown as soon as at least one **Plot** in the **Figure** has an **identifier**. A tool is free to choose both labeling scheme (such as *a*, *b*, ..., or *i*, *ii*, ...), placement in the plot, and styling in the plot itself as well as in other contexts.

When a **Figure** defines a non-empty **group**, it is used to organize figures similar to how **group** is used in the **Dialog** annotation (see section 18.8). However, leaving **group** at the default of an empty string does

not mean that a group will be created automatically, but that the figure resides outside of any group. The **group** is both the key used for grouping, and the name of the group for display purposes.

The **preferred** attribute of **Figure** indicates whether the figure should be given preference when automatically determining which figures to show, and a class may define any number of **preferred** figures. For example, a tool might choose to automatically show all preferred figures when the class is simulated.

The **caption** attribute of **Figure** can use the restricted form of text markup described in section 18.3.2.5 as well as the variable replacements described in section 18.3.2.4.

18.3.2.1 Axis Properties

Properties may be defined for each **Plot** axis:

```
record Axis
  /*literal*/ constant Real min "Axis lower bound, in 'unit'";
  /*literal*/ constant Real max "Axis upper bound, in 'unit'";
  /*literal*/ constant String unit = "" "Unit of axis tick labels";
  /*literal*/ constant String label "Axis label";
  /*literal*/ constant AxisScale scale = Linear() "Mapping between axis values
    and position on axis"
end Axis;
```

When an axis bound is not provided, the tool computes one automatically.

A non-empty **unit** shall match *unit-expression* in chapter 19. An empty **unit** means that the axis is unitless, and each expression plotted against it may use its own unit determined by the tool. The tool is responsible for conveying the information about choice of unit for the different variables, for instance by attaching this information to curve legends.

The Modelica tool is responsible for showing that values at the axis tick marks are expressed in **unit**, so the axis **label** shall not contain this information.

[*When* **unit** *is empty, and axis bounds are to be determined automatically, a natural choice of unit could be the variable's* **displayUnit**. *When axis bounds are specified by the user, on the other hand, a tool may choose a unit for the variable such that the range of the variable values (expressed in the chosen unit) fit nicely with the range of the unitless axis.*]

If a tool does not recognize the **unit**, it is recommended to issue a warning and treat the **unit** as if it was empty, as well as ignore any setting for **min** and **max**.

When **label** is not provided, the tool produces a default label. Providing the empty string as **label** means that no label should be shown. Variable replacements, as described in section 18.3.2.4, can be used in **label**. The Modelica tool is responsible for showing the unit used for values at the axis tick marks, so the axis **label** shall not contain the unit.

The type of **scale** is defined as an empty partial record:

```
partial record AxisScale
end AxisScale;
```

The standardized annotations extending from **AxisScale** are **Linear** and **Log**, but it is also allowed to use a vendor-specific annotation.

Use **Linear** for a linear mapping between axis values and position on axis:

```
record Linear
  extends AxisScale;
end Linear;
```

Use **Log** for a logarithmic mapping between axis values and position on axis:

```
record Log
  extends AxisScale;
  /*literal*/ constant Integer base(min = 2) = 10;
end Log;
```

The **base** of a **Log** scale determines preferred positions of major axis ticks. It is not required that the presentation of axis tick labels reflect the **base** setting. For example, when **base** is 10, major axis ticks should preferrably be placed at integer powers of 10, and natural alternatives that a tool may use for major axis tick labels could look like 0.001 or 10^{-3}. Under some circumstances, such as when the axis range does not span even a single order of magnitude, a tool may disregard the preference in order to get useful axis ticks.

[*Example: A* symmetric log *axis scale is sometimes used for axes spanning across several orders of magnitude of both positive and negative values. Details vary, but the mapping from value to linear position along axis is some variation of* $y \mapsto \text{sign}(y)\log(1+\frac{|y|}{10^{\alpha}})$. *A tool may implement this as a vendor-specific axis scale:*

```
Axis(
  min = -1e5, max = 1e5,
  scale = __NameOfVendor_symlog(1),
)
```

]

18.3.2.2 Plot Curves

The actual data to plot is specified in the **curves** of a **Plot**:

```
record Curve
  expression x = time "X coordinate values";
  expression y "Y coordinate values";
  /*literal*/ constant String legend "Legend";
  /*literal*/ constant Integer zOrder = 0 "Drawing order control";
end Curve;
```

The mandatory **x** and **y** expressions are restricted to be result references in the form of ***result-reference*** in the grammar (appendix A.2.7), referring to a scalar variable (or a derivative thereof) or **time**. It is an error if **x** or **y** does not designate a scalar result. If **x** or **y** is a derivative, `der(v, n)`, then n must not exceed the maximum amount of differentiation applied to **v** in the model. A diagnostic is recommended in case the simulation result is missing a trajectory for a valid result reference.

[*While the syntax for referring to a second order derivative is* `der(v, 2)`*, the appearance is left for tools to decide. For example, a tool might choose to present this as* `der(der(v))`.]

When the **unit** of an **Axis** is non-empty, it is an error if the unit of the corresponding **x** or **y** expression (i.e., a variable's **unit**, or second for **time**) is incompatible with the axis unit.

When **legend** is not provided, the tool produces a default based on **x** and/or **y**. Providing the empty string as **legend** means that the curve shall be omitted from the plot legend. Variable replacements, as described in section 18.3.2.4, can be used in **legend**. The order of presentation within the plot legend corresponds to order of appearance in the **curves** of a **Plot**.

The **zOrder** gives control over drawing order, with higher values corresponding to closer to front. Ties are resolved using order of appearance in the **curves** of a **Plot**, with later appearance corresponding to closer to front.

18.3.2.3 Escape Sequences

In an attribute inside a figure where the variable replacements of section 18.3.2.4 or the text markup of section 18.3.2.5 can be used, the following use of *text markup escape sequences* applies. These escape sequences are applied after the application of other markup, and is not applied at all inside some of the other markup, see details for the respective markup.

The percent character '%' shall be encoded **%%**. The following are all the recognized escape sequences:

Sequence	*Encoded character*	*Comment*
%%	'%'	Only way to encode character.
%]	']'	Prevents termination of markup delimited by `[...]`.

[*With the percent character being encoded as* `%%`*, the behavior of* `%` *appearing in any other way than the escape sequences above, for variable replacement (section 18.3.2.4), or for the text markup (section 18.3.2.5) is undefined, and thus possible to define in the future without breaking backward compatibility.*]

18.3.2.4 Variable Replacements

In the places listed in table 18.1 where text for display is defined, the final value of a result variable can be embedded by referring to the variable as `%{inertia1.w}`. This is similar to the `Text` graphical primitive in section 18.7.5.5.

Table 18.1: *Attributes that can use variable replacements.*

Attribute	*Annotation*
`title`	`Figure` and `Plot`
`caption`	`Figure`
`legend`	`Curve`
`label`	`Axis`

In `%{`*variable*`}`, text markup escape sequences don't apply inside the *variable*, which has the form of ***`result-reference`***. This means that a complete ***`result-reference`*** shall be scanned before looking for the terminating closing brace.

[*Example: The variable replacement* `%{'%%'}` *references the variable* `'%%'`*, not the variable* `'%'`.]

[*Example: The variable replacement* `%{foo . '}bar{'}` *makes a valid reference to the variable* `foo.'}bar{'`.]

Note that expansion to the final value means that expansion is not restricted to parameters and constants, so that values to be shown in a caption can be determined during simulation.

[*By design, neither* `%class` *nor* `%name` *is supported in this context, as this information is expected to already be easily accessible (when applicable) in tool-specific ways. (Titles making use of* `%class` *or* `%name` *would then only lead to ugly duplication of this information.)*]

18.3.2.5 Text Markup in Captions

In addition to variable replacements, a very restricted form of text markup is used for the `caption`. Note that the text markup escape sequences described in section 18.3.2.3 generally apply inside `caption`, with one exception given below for links.

Links take the form `%[`*text*`](`*link*`)`, where the `[`*text*`]` part is optional, and text markup escape sequences don't apply inside the *link*. The *link* can be in either of the following forms, where the interpretation is given by the first matching form:

- A `variable:`*id*, where *id* is a component reference in the form of ***`result-reference`*** in the grammar, such as `inertia1.w`.
- A `plot:`*id*, where *id* is the identifier of a `Plot` in the current `Figure`.
- A URI. Well established schemes such as `https://github.com/modelica` or `modelica:/Modelica` , as well as lesser known schemes may be used. (A tool that has no special recognition of a scheme can try sending the URI to the operating system for interpretation.)

When `[`*text*`]` is omitted, a Modelica tool is free to derive a default based on the *link*.

[*Note that for the character '*`]`*' to appear in text, it needs to be encoded as the escape sequence* `%]`*, or it would be interpreted as the terminating delimiter of the* `[`*text*`]`.

Similarly, the closing parenthesis '`)`*' must be handled with care in link in order to not be interpreted as the terminating delimiter of the* `(`*link*`)`.

- *For a* `variable:`*, no special treatment is needed, as the component reference syntax of the id allows parentheses to appear without risk of misinterpretation inside a quoted identifier. For example,* `%(`

`variable:'try)me!'`) *has a parenthesis in* `'try)me!'` *that must not be mistaken for the end of the* (*link*).

- *For a* `plot:`*, there is currently no way to reference a plot with ')' in its* `identifier`.
- *For a URI, a closing parenthesis must be URL encoded in order to not be interpreted as the end of the* (*link*). *For example, the URL in* `%(http://example.org/(tryme))` *is just http://example.org/(tryme, and the entire link is followed by a stray closing parenthesis. To make it work, one has to use URL encoding:* `%(http://example.org/%28tryme%29)` *(using URL encoding of the opening parenthesis just for symmetry, and note that the* `%` *of the percent-encoded sequences are not subject to text markup escape sequences).*

]

The styling of the link text, as well as the link action, is left for each Modelica tool to decide.

[*For example,* `%(variable:inertia1.w)` *could be displayed as the text* `inertia1.w` *formatted with upright monospaced font, and have a pop-up menu attached with menu items for plotting the variable, setting its start value, or investigating the equation system from which it is solved. On the other hand,* `%[angular velocity](variable:inertia1.w)` *could be formatted in the same style as the surrounding text, except some non-intrusive visual clue about it being linked.*]

[*Note that link is currently not allowed to be a* URI reference*, i.e., a URI or a* relative reference *such as* `#foo`*. This is due to to the current inability to define a base URI referencing the current figure. Once this becomes possible, the URI form of link may be changed into a URI reference.*]

A sequence of one or more newlines (encoded either literally or using the `\n` escape sequence) means a paragraph break. (A line break within a paragraph is not supported, and any paragraph break before the first paragraph or after the last paragraph has no impact.)

Vendor-specific markup takes the form `%__`*nameOfVendor*$_1$`(`*data*$_1$`)...__`*nameOfVendor*$_n$`(`*data*$_n$`)[`*text*`]`, where $n \geq 1$. The *nameOfVendor* consists of only digits and letters, and shall only convey the name of the vendor defining the meaning of the associated *data*. Text markup escape sequences don't apply inside the *data*, implying that it cannot contain the closing parenthesis, ')'. A tool which does not understand any of the vendor-specific meanings shall only display the mandatory *text*, but the *text* may also be used together with the vendor-specific *data*.

[*Example: One application of vendor-specific markup is to prototype a feature that can later be turned into standardized markup. For example, say that the tool AVendor wants to generalize the variable replacements such that the duration of a simulation can be substituted into a caption. During the development, this could be represented as the vendor-specific markup* `%__AVendor(?duration)[10 s]`*, if the simulation has a duration of 10 seconds at the time of writing the caption. When AVendor renders this, it ignores the text* `10 s` *and just displays the actual duration instead. Later, if this would become supported by standard markup, it might take the form of something like* `%{experiment:duration}` *instead (note that* `experiment:duration` *is not in the form of a component reference, avoiding conflict with current use of variable replacements).*

In a similar way, vendor-specific markup can be used to prototype a link for future inclusion in the link markup (either by extending the meaning of Modelica URIs, or by introducing another pseudo-scheme similar to `variable:`*). This is an example where the vendor-specific markup could make use of the text (for link text) together with the vendor-specific data (describing the actual link).*]

18.4 Symbolic Processing

The annotation listed below, in addition to annotations described in sections 12.7 to 12.8, can influence the symbolic processing.

Annotation	*Description*	*Details*
`Evaluate`	Use parameter value for symbolic processing	Annotation 18.1

Annotation 18.1 `Evaluate`

```
/*literal*/ constant Boolean Evaluate;
```

The annotation `Evaluate` can occur in the component declaration, its type declaration, or a base class of the type-declaration. In the case of multiple conflicting annotations it is handled similarly to modifiers (e.g., an `Evaluate` annotation on the component declaration takes precedence). In the case of hierarchical components it is applied to all components, overriding any `Evaluate`-setting for specific components. The annotation `Evaluate` is only allowed for parameters and constants.

Setting `Evaluate = true` for an evaluable parameter, means that it must be an evaluated parameter (i.e., its value must be determined during translation, similar to a constant). For a non-evaluable parameter, it has no impact and it is recommended to issue a warning in most cases. The exception for recommending this warning is when the parameter is non-evaluable due to dependency on a parameter with `Evaluate = false`, as this could be a sign of intentional overriding of `Evaluate = true`, see example below. For both evaluable parameters and constants, the model developer further proposes to utilize the value for symbolic processing. A constant can never be changed after translation, and it is normal for its value to be used for symbolic processing even without `Evaluate = true`.

For a parameter, `Evaluate = false` ensures that the parameter is a non-evaluable parameter according to section 4.5 (meaning it is not allowed to be used where an evaluable expression (section 3.8.3) is expected). For both parameters and constants – even when the value can be determined during translation – the model developer further proposes to not utilize the value for symbolic processing.

[`Evaluate = true` *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.*

Conversely, a possible use of `Evaluate = false` *is to ensure that a parameter can be changed after translation, even when a tool might be tempted to evaluate it to improve the efficiency of the generated code.*]

[*Example: When a parameter has* `Evaluate = true` *for optimization reasons (not because it needs to be evaluable), it is possible to prevent the value from being determined during translation without modifying the original model:*

```
model M_evaluable
  /* Here, 'b' is evaluable, and will be evaluated. */
  parameter Boolean b = false annotation(Evaluate = true);
  Real x(start = 1.0, fixed = true);
equation
  if b then /* No need for b to be evaluable. */
    der(x) = x;
  else
    der(x) = -x;
  end if;
end M_evaluable;

model M_non_evaluable
  /* Here, 'bn' is non-evaluable, which in turn will cause 'b' to be
   * non-evaluable, thereby preventing it from being determined during
   * translation.
   */
  extends M_evaluable(b = bn);
  parameter Boolean bn = false annotation(Evaluate = false);
end M_non_evaluable;
```

]

18.5 Simulations

The annotations listed below define how models can be checked, translated, and simulated.

Annotation	*Description*	*Details*
`experiment`	Simulation experiment settings	Annotation 18.2
`HideResult`	Don't show component's simulation result	Annotation 18.3
`TestCase`	Information for model used as test case	Annotation 18.4

Annotation 18.2 `experiment`

```
record experiment
  /*literal*/ constant Real StartTime(unit = "s") = 0;
  /*literal*/ constant Real StopTime(unit = "s");
  /*literal*/ constant Real Interval(unit = "s");
  /*literal*/ constant Real Tolerance(unit = "1");
end experiment;
```

The `experiment` annotation defines the start time (`StartTime`) in [s], the stop time (`StopTime`) in [s], the suitable time resolution for the result grid (`Interval`) in [s], and the relative integration tolerance (`Tolerance`) for simulation experiments to be carried out with the model or block at hand. When `Interval` or `Tolerance` is not provided, the tool is responsible for applying appropriate defaults.

The experiment options are inherited, and the derived class may override individual inherited options.

[*The inheritance makes it useful to have an* `experiment` *annotation also in partial models, e.g., a template for a number of similar test cases.*]

If `StopTime` is set in a non-partial model, it is required to be a simulation model. Tools can allow users to override these settings without modifying the model.

Annotation 18.3 `HideResult`

```
/*literal*/ constant Boolean HideResult;
```

`HideResult = true` defines that the model developer proposes to not show the simulation results of the corresponding component.

`HideResult = false` defines that the developer proposes to show the corresponding component.

[*For example, a tool is not expected to provide means to plot a variable with* `HideResult = true`*. 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.*

`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.*]

Annotation 18.4 `TestCase`

```
record TestCase
  /*literal*/ constant Boolean shouldPass;
end TestCase;
```

If `shouldPass` is `false` it indicates that the translation or the simulation of the model should fail. If a tools checks a package where classes have `shouldPass = false` they should not generate errors, and checking may even be skipped. On the other hand, models with `shouldPass = false` may be useful for creation of negative tests in tool-specific ways. Similarly as a class with obsolete-annotation, a class with `TestCase` annotation (regardless of the value of `shouldPass`) shall not be used in other models, unless those models also have a `TestCase` annotation.

[*The intent of the test-case can be included in the documentation of the class. This annotation can both be used for models intended as test-cases for implementations, and for models explaining detectable errors.*]

18.6 Usage Restrictions

18.6.1 Single Use of Class

For state machines it is useful to have single instances of local classes. This can be done using:

```
/*literal*/ constant Boolean singleInstance;
```

The annotation **singleInstance**, if **true**, in a class indicates that there should only be one component instance of the class, and it should be in the same scope as the class is defined. The intent is to remove the class when the component is removed and to prevent duplication of the component. If **false**, it has no impact.

18.6.2 Connection Restrictions

A connector component declaration may have the following annotation:

```
/*literal*/ constant String mustBeConnected;
```

It makes it an error if the connector is not connected from the outside (for a conditional connector this check is only active if the connector is enabled). The string value should provide the reason why it must be connected. For an array of connectors it applies separately to each element.

[*This annotation is intended for non-causal connectors, see section 9.3. It is particularly suited for stream connectors, see chapter 15.*]

[*Example: This can be used for some optional connectors that should be connected when conditionally enabled.*

```
partial model PartialWithSupport
  Flange_b flange;
  parameter Boolean useSupport;
  Support support if useSupport
    annotation(
      mustBeConnected = "Support connector should be connected if activated.");
end PartialWithSupport;
```

The protected components and connections needed to internally handle the support-connector is omitted.]

A connector component declaration may have the following annotation:

```
/*literal*/ constant String mayOnlyConnectOnce;
```

It makes it an error if the connector is connected from the outside and:

- For non-stream connectors the connection set has more than two elements.
- For stream connectors (see chapter 15), the connection set has more than two elements whose flow variable may be negative (based on evaluation of the **min**-attribute).

For an array of connectors it applies separately to each element. The string value should provide the reason why it may only be connected once.

[*This annotation is intended for non-causal connectors, see section 9.3. The connection handling operates on connection sets, and thus this restriction should also operate on those sets. The set handling avoids the case where only one of two equivalent models generate diagnostics. The stream connector part is primarily intended to exclude sensor-variables, see appendix C.3.3, but also excludes non-reversible outgoing flows.*]

[*Example: This can be used for components that implement mixing of fluids where it is not desired to combine that with the normal stream-connector mixing.*

```
partial model MultiPort
  parameter Integer n = 0 annotation(Dialog(connectorSizing = true));
  FluidPort_a port_a(redeclare package Medium = Medium);
  FluidPorts_b ports_b[n](redeclare each package Medium = Medium)
    annotation (mayOnlyConnectOnce = "Should only connect once per element!");
end MultiPort;
```

]

18.7 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(extent = {{-100, -100}, {100, 100}},
                         textString = "Icon")}));
```

]

The graphics is specified as an ordered sequence of graphical primitives described below. Base class contents are drawn behind the graphical primitives of the current class, with base classes ordered from back to front according to the order of the **extends**-clauses, and graphical primitives according to order of appearance in the annotation.

[*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.*]

These `Icon`, `Diagram`, and `Documentation` annotations are only allowed directly in classes (e.g., not on components or connections). The allowed annotations for a short class definition is the union of the allowed annotations in classes and on **extends**-clauses.

18.7.1 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 counter-clockwise around the point defined by the **origin** attribute.

18.7.1.1 Coordinate Systems

Each of the layers has its own coordinate system. A coordinate system is defined by the coordinates of two points, the left (x1) lower (y1) corner and the right (x2) upper (y2) corner, where the coordinates of the first point shall be less than the coordinates of the second point.

The attribute **preserveAspectRatio** specifies a hint for the shape of components of the class, but does not actually influence the rendering of the component. If **preserveAspectRatio** is true, changing the extent of components should preserve the current aspect ratio of the coordinate system of the class.

The attribute **initialScale** specifies the default component size as **initialScale** times the size 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
  /*literal*/ constant Extent extent;
  /*literal*/ constant Boolean preserveAspectRatio = true;
  /*literal*/ constant Real initialScale = 0.1;
  /*literal*/ constant 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 **GraphicItem** 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;
```

The coordinate system attributes (**extent** and **preserveAspectRatio**) of a class are separately defined by the following priority:

1. The coordinate system annotation given in the class (if specified).
2. The coordinate systems of the first base class where the extent on the **extends**-clause specifies a null-region (if any). Note that null-region is the default for base classes, see section 18.7.3.
3. The default coordinate system `CoordinateSystem(preserveAspectRatio=true, extent = {{-100, -100}, {100, 100}})`.

18.7.1.2 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);
type EllipseClosure = enumeration(None, Chord, Radial);
```

The `LinePattern` attribute `Solid` indicates a normal line, `None` an invisible line, and the other attributes various forms of dashed/dotted lines.

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 — inside the extent of the filled shape.

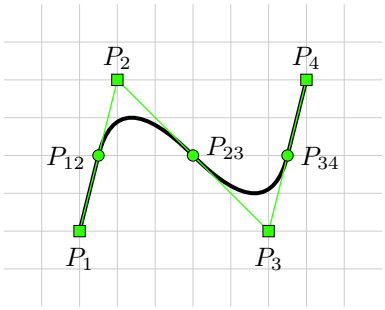


Figure 18.1: *Line with* `smooth = Bezier`. *The four line points $P_1, \ldots, P_4$ result in two quadratic splines and two straight line segments.*

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, see figure 18.1.

For lines with only two points, the `smooth` attribute has no effect.

For lines with three or more points (P_1, P_2, ..., P_n), the middle point of each line segment (P_{12}, P_{23}, ..., $P_{(n-1)n}$) 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.

The values of the `EllipseClosure` enumeration specify if and how the endpoints of an elliptical arc are to be joined (see section 18.7.5.4).

```
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;
```

The extent/points of the filled shape describe the theoretical zero-thickness filled shape, and the actual rendered border is then half inside and half outside the extent.

18.7.2 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 terms of coordinate system transformations:

```
record Transformation
  Extent extent;
  Real rotation(quantity = "angle", unit = "deg") = 0;
  Point origin = {0, 0};
end Transformation;
```

The attributes are applied in the order **`extent`**, **`rotation`**, **`origin`**, as follows:

1. The **`extent`** of the component icon is mapped to the **`extent`** rectangle (possibly shifting, scaling, and flipping contents).
2. The **`rotation`** specifies counter-clockwise rotation around the origin (that is `{0, 0}`, not the **`origin`** attribute).
3. The **`origin`** specifies a shift (moving `{0, 0}` to **`origin`**).

```
record Placement
  Boolean visible = true;
  Transformation transformation "Placement in the diagram layer";
  Boolean iconVisible "Visible in icon layer; for public connector";
  Transformation iconTransformation
    "Placement in the icon layer; for public connector";
end Placement;
```

If no `iconTransformation` is given the `transformation` is also used for placement in the icon layer. If no `iconVisible` is given for a public connector the `visible` is also used for visibility in the icon layer.

[*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.

18.7.3 Extends-Clause

Each **`extends`**-clause (and short class definition, as stated in section 18.7) may have layer specific annotations which describe the rendering of the base class' icon and diagram layers in the derived class.

```
record IconMap
  /*literal*/ constant Extent extent = {{0, 0}, {0, 0}};
  /*literal*/ constant Boolean primitivesVisible = true;
end IconMap;

record DiagramMap
  /*literal*/ constant Extent extent = {{0, 0}, {0, 0}};
  /*literal*/ constant Boolean primitivesVisible = true;
end DiagramMap;
```

All graphical objects are by default inherited from a base class. If the **primitivesVisible** attribute is false, components and connections are visible but graphical primitives are not.

- If the **extent** is **{{0, 0}, {0, 0}}** (the default), the base class contents is mapped to the same coordinates in the derived class, and the coordinate system (including **preserveAspectRatio**) can be inherited as described in section 18.7.1.1.
- For any other **extent**, the base class coordinate system is mapped to this region, with the exception that **preserveAspectRatio** = **true** in the base class requires that the mapping shall preserve the aspect ratio. The base class coordinate system (and **preserveAspectRatio**) is not inherited.

[*A zero area* **extent** *other than* **{{0, 0}, {0, 0}}** *will result in none of the base class contents being visible. By affecting components and connections as well as graphical primitives, this is different from setting* **primitivesVisible** = **false**.

Reversed corners of the **extent** *will result in mirrored (rotated if reversed in both direction) base class contents.*]

[*Example:*

```
model A
  extends B annotation(
    IconMap(extent = {{-100, -100}, {100, 100}}, primitivesVisible = false),
    DiagramMap(extent = {{-50, -50}, {0, 0}}, primitivesVisible = true)
  );
end A;

model B
  extends C annotation(DiagramMap(primitivesVisible = false));
  ...
end B;
```

In this example the diagram of **A** *contains the graphical primitives from* **A** *and* **B** *(but not from* **C** *since they were hidden in* **B***) – the ones from* **B** *are rescaled, and the icon of* **A** *contains the graphical primitives from* **A** *(but neither from* **B** *nor from* **C***).*]

18.7.4 Connections

A connection is specified with an annotation containing a **Line** primitive and optionally a **Text** primitive, as specified below.

[*Example:*

```
connect(a.x, b.x)
  annotation(Line(points = {{-25, 30}, {10, 30}, {10, -20}, {40, -20}}));
```

]

The optional **Text** primitive defines a text that will be written on the connection line. It has the following definition (it is not equal to the **Text** primitive as part of graphics – the differences are marked after *Note* in the description-strings):

```
record Text
  extends GraphicItem;
  Extent extent;
```

```
  String string "Note: different name";
  Real fontSize = 0 "unit pt";
  String fontName;
  TextStyle textStyle[:];
  Color textColor = Black;
  TextAlignment horizontalAlignment =
    if index < 0 then TextAlignment.Right else TextAligment.Left "Note:
    different default";
  Integer index "Note: new";
end Text;
```

The **index** is one of the points of Line (numbered 1, 2, 3, ... where negative numbers count from the end, thus -1 indicate the last one). The **string** may use the special symbols **"%first"** and **"%second"** to indicate the connectors in the **connect**-equation.

The **extent** and **rotation** are relative to the **origin** (default **{0, 0}**) and the **origin** is relative to the point on the **Line**.

The **textColor** attribute defines the color of the text. The text is drawn with transparent background and no border around the text (and without outline). The contents inherited from **FilledShape** is deprecated, but kept for compatibility reasons. The default value for **horizontalAlignment** is deprecated. Having a zero size for the **extent** is deprecated and is handled as if upper part is moved up an appropriate amount.

[*Example:*

```
connect(controlBus.axisControlBus1, axis1.axisControlBus)
  annotation(
    Text(string = "%first", index = -1, extent = [-6, 3; -6, 7]),
    Line(points = {{41, 30}, {50, 30}, {50, 50}, {58, 50}})
  );
```

Draws a connection line and adds the text axisControlBus1 *ending at* $(-6, 3) + (58, 50)$ *and 4 vertical units of space for the text. Using a height of zero, such as* **extent = [-6, 3; -6, 3]** *is deprecated, but gives similar result.*]

18.7.5 Graphical Primitives

This section describes the graphical primitives that can be used to define the graphical objects in an annotation.

18.7.5.1 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.

For arrows:

- The arrow is drawn with an aspect ratio of 1/3 for each arrow half, i.e., if the arrow-head is 3 mm long an arrow with **Half** will extend 1 mm from the mid-line and with **Open** or **Filled** extend 1 mm to each side, in total making the base 2 mm wide.

- The `arrowSize` gives the width of the arrow (including the imagined other half for `Half`) so that `lineThickness = 10` and `arrowSize = 10` will touch at the outer parts.
- All arrow variants overlap for overlapping lines.
- The lines for the `Open` and `Half` variants are drawn with `lineThickness`.

18.7.5.2 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.

18.7.5.3 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.

18.7.5.4 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;
  EllipseClosure closure =
    if startAngle == 0 and endAngle == 360 then
      EllipseClosure.Chord
    else
      EllipseClosure.Radial;
end Ellipse;
```

The `extent` attribute specifies the bounding box of the ellipse.

Partial ellipses can be drawn using the `startAngle` and `endAngle` attributes. These specify the endpoints of 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 (the positive x-axis).

The closure attribute specifies whether the endpoints specified by `startAngle` and `endAngle` are to be joined by lines to the center of the extent (`closure = EllipseClosure.Radial`), joined by a single straight line between the end points (`closure = EllipseClosure.Chord`), or left unconnected (`closure = EllipseClosure.None`). In the latter case, the ellipse is treated as an open curve instead of a closed shape, and the `fillPattern` and `fillColor` are not applied (if present, they are ignored).

The default closure is `EllipseClosure.Chord` when `startAngle` is 0 and `endAngle` is 360, or `EllipseClosure.Radial` otherwise.

[*The default for a closed ellipse is not* `EllipseClosure.None`*, since that would result in* `fillColor` *and* `fillPattern` *being ignored, making it impossible to draw a filled ellipse.* `EllipseClosure.Chord` *is equivalent in this case, since the chord will be of zero length.*]

18.7.5.5 Text

A text string is specified as follows:

```
record Text
  extends GraphicItem;
  Extent extent;
  String textString;
  Real fontSize = 0 "unit pt";
  String fontName;
  TextStyle textStyle[:];
  Color textColor = Black;
  TextAlignment horizontalAlignment = TextAlignment.Center;
end Text;
```

The `textColor` attribute defines the color of the text. The text is drawn with transparent background and no border around the text (and without outline).

There are a number of common macros that can be used in the text, and they should be replaced when displaying the text as follows (in order such that the earliest ones have precedence, and using the longest sequence of identifier characters – alphanumeric and underscore):

- %% replaced by %
- %name replaced by the name of the component (i.e., the identifier for it in the enclosing class).
- %class replaced by the name of the class (only the last part of the hierarchical name).
- %*par* and %{*par*} replaced by the value of the parameter `par`. If the value is numeric, tools shall display the value with `displayUnit`, formatted according to the BIPM specification. E.g., for

  ```
  parameter Real t(unit = "s", displayUnit = "ms") = 0.1
  ```

  tools shall display *100 ms*. The intent is that the text is easily readable, thus if `par` is of an enumeration type, replace %*par* by the item name, not by the full name.

  [*Example: If* `par = "Modelica.Blocks.Types.Enumeration.Periodic"`*, then* %par *should be displayed as* Periodic.]

  When quoted identifiers (e.g., `rec.'}'.'quoted ident'`) are involved, the form %{*par*} must be used. Here, *par* is a general *`component-reference`*, and the macro can be directly followed by a letter. Thus `%{w}x%{h}` gives the value of `w` directly followed by *x* and the value of `h`, while `%wxh` gives the value of the parameter `wxh`. If the parameter does not exist it is an error.

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 `extent` specifies a box with zero width and positive height the height is used as height for the text (unless `fontSize` attribute is non-zero – which specifies the absolute size), and the text is not truncated (the `horizontalAlignment` is still used in this case).

[*A zero-width* `extent` *is convenient for handling texts where the width is unknown.*]

If the string `fontName` is empty, the tool may choose a font. The font names `"serif"`, `"sans-serif"`, and `"monospace"` shall be recognized. If possible the correct font should be used – otherwise a reasonable match, or treat as if `fontName` was empty.

The style attribute `textStyle` specifies variations of the font.

18.7.5.6 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. 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 center of the object.

The graphical operations are applied in the order: scaling, flipping and rotation.

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 specified for some URIs in section 13.5. The supported file formats include `PNG`, `BMP`, `JPEG`, and `SVG`.

When the attribute `imageSource` is specified, the string contains the image data, and the image format is determined based on the contents. 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 (preserving the 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.

18.7.6 Variable Graphics and Schematic Animation

Any value (coordinates, color, text, etc.) in graphical annotations can be dependent on class variables using `DynamicSelect`. `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})
  )}));
```

]

18.7.7 User Input

It is possible to interactively modify variables during a simulation. The variables may either be parameters, discrete-time 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.

18.7.7.1 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 OnMouseDownSetBoolean;
```

[*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 = {OnMouseDownSetBoolean(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 OnMouseUpSetBoolean;
```

Note that several interaction objects can be associated with the same graphical item or component.

[*Example:*

```
interaction = {OnMouseDownSetBoolean(on, true),
               OnMouseUpSetBoolean(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;
```

18.7.7.2 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;
```

18.8 Graphical User Interface

This section describes the annotations that are used to define properties of the graphical user interface.

```
/*literal*/ constant String preferredView = view;
```

The **preferredView** annotation defines the default view when selecting the class. The *view* is a String literal where **"info"** means class documentation ("information"), **"diagram"** means diagram view, **"icon"** means icon view, and **"text"** means Modelica source code ("text").

```
/*literal*/ constant Boolean DocumentationClass;
```

Only allowed as class annotation on any kind of class and, if set to true, implies that this class and all classes within it are treated as having the annotation **preferredView** = **"info"**. If the annotation **preferredView** is explicitly set for a class, it has precedence over a **DocumentationClass** annotation.

[*A tool may display such classes in special ways. For example, the description texts of the classes might be displayed instead of the class names, and if no icon is defined, a special information default icon may be displayed in the package browser.*]

```
/*literal*/ constant String defaultComponentName = "name";
```

When creating a component of the given class, the recommended component name is *name*. If the default name cannot be used (e.g., since it is already in use), another name based on **defaultComponentName** shall be derived automatically, except as described under **defaultComponentPrefixes**. When automatically deriving a name, any trailing '1' in the **defaultComponentName** shall be disregarded.

```
/*literal*/ constant String defaultComponentPrefixes = "prefixes";
```

When creating a component, it is recommended to generate a declaration of the form

```
type-prefix type-specifier component-declaration
```

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. If the prefixes contain* **inner** *or* **outer** *and the default name cannot be used (e.g., since it is already in use) it is recommended to give a diagnostic.*]

```
/*literal*/ constant String missingInnerMessage = "message";
```

When an **outer** component of the class does not have a corresponding **inner** component, the literal string message may be used as part of a diagnostic message (together with appropriate context), see section 5.4.

[*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 simple type or component of a simple type may have:

```
/*literal*/ constant Boolean absoluteValue;
```

If `false`, then the variable defines a relative quantity, and if `true` an absolute quantity.

[*When converting between units (in the user-interface for plotting and entering parameters), the unit offset must be ignored for a variable defined with annotation* `absoluteValue = false`. *This annotation is used in the Modelica Standard Library, for example in* `Modelica.Units.SI` *for the type definition* `TemperatureDifference`.]

A model or block definition may contain:

```
/*literal*/ constant Boolean defaultConnectionStructurallyInconsistent;
```

If `true`, it is stated that a default connection will result in a structurally inconsistent model or block[1]. A "default connection" is constructed by instantiating the respective **`model`** or **`block`** and for every input `u` providing an equation `0 = f(u)`, and for every (potential, flow) pair of the form `(v, i)`, providing an equation of the form `0 = f(v, i)`.

[*It is useful to check all models/blocks of a Modelica package in a simple way. One check is to default connect every model/block and to check whether the resulting class is structurally consistent (which is a stronger requirement than being balanced). It is rarely needed; but is for example used in* `Modelica.Blocks.Math.InverseBlockConstraints`, *in order to prevent a wrong error message. Additionally, when a user defined model is structurally inconsistent, a tool should try to pinpoint in which class the error is present. This annotation avoids then to show a wrong error message.*]

A class may have the following annotation:

```
/*literal*/ constant String obsolete = "message";
```

It indicates that the class ideally should not be used anymore and gives a message indicating the recommended action. This annotation is not inherited, the assumption is that if a class uses an obsolete class (as a base class or as the class of one of the components) that shall be updated – ideally without impacting users of the class. If that is not possible the current class can have also have an **`obsolete`** annotation.

A component declaration may have the following annotation:

```
/*literal*/ constant String 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` *or* `b = -a` *cannot be assigned, the message is displayed. This annotation is used to provide library specific error messages.*]

[*Example:*

```
connector Frame "Frame of a mechanical system"
  ...
  flow Modelica.Units.SI.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
```

[1] For the precise definition of *structurally inconsistent*, see Pantelides (1988).

```
    PlanarCutJoint=true in the Advanced menu.
    ");
end Frame;
```

]

A component declaration or a short replaceable class definition may have the following annotation:

```
record Dialog
  /*literal*/ constant String tab = "General";
  /*literal*/ constant String group = "";
  /*evaluable*/ parameter Boolean enable = true;
  /*literal*/ constant Boolean showStartAttribute = false;
  /*literal*/ constant Boolean colorSelector = false;
  /*literal*/ constant Selector loadSelector;
  /*literal*/ constant Selector saveSelector;
  /*literal*/ constant Selector directorySelector;
  /*literal*/ constant String groupImage = "";
  /*literal*/ constant Boolean connectorSizing = false;
end Dialog;

record Selector
  /*literal*/ constant String filter = "";
  /*literal*/ constant String caption = "";
end Selector;
```

For a short replaceable class definition only the fields `tab`, `group`, `enable` and `groupImage` are allowed.

In the organization of a tool's user interface, the `tab` shall correspond to a major divisioning of "tabs", and `group` correspond to sub-divisioning of "groups" within each tab. An empty `group` (the default) means tool-specific choice of group. The order of components (and class definitions) within each group and the order of the groups and tabs are according to the declaration order, where inherited elements are added at the place of the extends.

A component shall have at most one of `showStartAttribute=true`, `colorSelector=true`, `loadSelector`, `saveSelector`, `directorySelector`, or `connectorSizing=true`.

[*Example: When* `group` *is empty, a tool may place parameters in the group "Parameters", and place variables with* `showStartAttribute = true` *in the group "Start Attributes".*]

If `enable = false`, the input field may be disabled and no input can be given.

If `showStartAttribute = true` the dialog should allow the user to set the `start`- and `fixed`-attributes for the variable instead of the value of the variable.

[*The* `showStartAttribute = true` *is primarily intended for non-parameter values and avoids introducing a separate parameter for the* `start`*-attribute of the variable.*]

If `colorSelector = true`, it suggests the use of a color selector to pick an RGB color as a vector of three values in the range 0..255 (the color selector should be useable both for vectors of `Integer` and `Real`).

The presence of `loadSelector` or `saveSelector` specifying `Selector` suggests the use of a file dialog to select a file. Setting `filter` will in the dialog only show files that fulfill the given pattern. Setting `text1 (*.ext1);;text2 (*.ext2)` will only show files with file extension `ext1` or `ext2` with the corresponding description texts `text1` and `text2`, respectively. `caption` is a caption for display in the file dialog. `loadSelector` is used to select an existing file for reading, whereas `saveSelector` is used to define a file for writing.

The presence of `directorySelector` specifying `Selector` suggests the use of a dialog to select an existing directory. The selected directory does not need to exist at the time of opening the dialog; it is allowed to let the dialog be used to create directory before selecting it. The `filter` may not be used. The `caption` is a caption for display in the file dialog.

The `groupImage` references an image using an URI (see section 13.5), and the image is intended to be shown together with the entire group (only one image per group is supported). Disabling the input

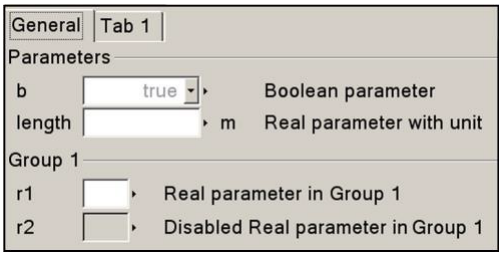
field will not disable the image. The background of the `groupImage` and any image used in HTML-documentation is recommended to be transparent (intended to be a light color) or white.

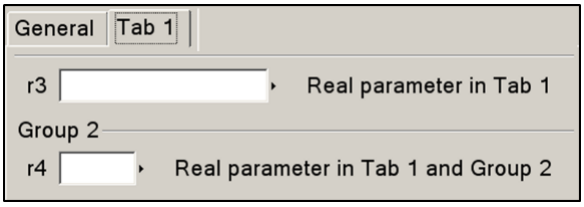
The `connectorSizing` is described separately in section 18.8.1.

[*Example:*

```
model DialogDemo
  parameter Boolean b = true "Boolean parameter";
  parameter Modelica.Units.SI.Length length "Real parameter with unit";
  parameter Real r1 "Real parameter in Group 1"
     annotation(Dialog(group = "Group 1"));
  parameter Real r2 "Disabled Real parameter in Group 1"
     annotation(Dialog(group = "Group 1", enable = not b));
  parameter Real r3 "Real parameter in Tab 1"
     annotation(Dialog(tab = "Tab 1"));
  parameter Real r4 "Real parameter in Tab 1 and Group 2"
     annotation(Dialog(tab = "Tab 1", group = "Group 2"));
  ...
end DialogDemo;
```

When clicking on an instance of model `DialogDemo`*, a dialog is shown that may have the following layout (other layouts are also possible, this is vendor specific).*





]

18.8.1 Connector Sizing

This section describes the `connectorSizing` annotation inside a `Dialog` annotation. The value of `connectorSizing` must be a literal `false` or `true`. If `connectorSizing = false`, this annotation has no effect. If `connectorSizing = true`, the corresponding variable must be declared with the `parameter` prefix, must be a subtype of a scalar `Integer` and must have a literal default value of zero.

[*The reason why* `connectorSizing` *must be given a literal value is that if the value is an expression, the* `connectorSizing` *functionality is conditional and this will then lead easily to wrong models.*

The default value of the variable must be zero since this annotation is designed for a parameter that is used as vector dimension, and the dimension of the vector should be zero when the component is dragged or redeclared. Furthermore, when a tool does not support the `connectorSizing` *annotation, dragging will still result in a correct model.*]

If `connectorSizing = true`, a tool may set the parameter value in a modifier automatically, if used as dimension size of a vector of connectors. In that case the parameter should not be modified by the user, and a tool may choose to not display that parameter in the dialog or display it with disabled input field.

[*The* `connectorSizing` *annotation is used in cases where connections to a vector of connectors shall be made and a new connection requires to resize the vector and to connect to the new index (unary connections). The annotation allows a tool to perform these two actions in many cases automatically. This is, e.g., very useful for state machines and for certain components of fluid libraries.*]

[*The following part is non-normative text and describes a useful way to handle the* `connectorSizing` *annotation in a tool (still a tool may use another strategy and/or may handle other cases than described below). The recommended rules are clarified at hand of the following example which represents a connector and a model from the* `Modelica.StateGraph` *library (note that they may be modified or renamed in future versions):*

```
connector Step_in // Only 1:1 connections are possible since input used
  output Boolean occupied;
  input Boolean set;
end Step_in;

block Step
  // nIn cannot be set through the dialog (but maybe shown)
  parameter Integer nIn=0 annotation(Dialog(connectorSizing=true));
  Step_in inPorts[nIn];
  ...
end Step;
```

If the parameter is used as dimension size of a vector of connectors, it is automatically updated according to the following rules:

1. *If a new connection line is drawn between one outside and one inside vector of connectors both dimensioned with (*connectorSizing*) parameters, a connection between the two vectors is performed and the (*connectorSizing*) parameter is propagated from connector to component. Other types of outside connections do not lead to an automatic update of a (*connectorSizing*) parameter.* Example: *Assume there is a connector* inPorts *and a component* step1*:*

   ```
   parameter Integer nIn=0 annotation(Dialog(connectorSizing=true));
   Step_in inPorts[nIn];
   Step step1(nIn=0);
   ```

   Drawing a connection line between connectors inPorts *and* step1.inPorts *results in:*

   ```
     parameter Integer nIn=0 annotation(Dialog(connectorSizing=true));
     Step_in inPorts[nIn];
     Step step1(nIn=nIn); // nIn=0 changed to nIn=nIn
   equation
     connect(inPorts, step1.inPorts); // new connect-equation
   ```

2. *If a connection line is deleted between one outside and one inside vector of connectors both dimensioned with (*connectorSizing*) parameters, the* connect*-equation is removed and the (*connectorSizing*) parameter of the component is set to zero or the modifier is removed.* Example: *Assume the connection line in the resulting example in case 1 is removed. This results in:*

   ```
   parameter Integer nIn=0 annotation(Dialog(connectorSizing=true));
   Step_in inPorts[nIn];
   Step step1; // modifier nIn=nIn is removed
   ```

3. *If a new connection line is drawn to an inside connector with* connectorSizing *and case 1 does not apply then, the parameter is incremented by one and the connection is performed for the new highest index.* Example: *Assume that 3 connections are present and a new connection is performed. The result is:*

   ```
     Step step1(nIn=4); // index changed from nIn=3 to nIn=4
   equation
     connect(..., step1.inPorts[4]); // new connect-equation
   ```

   In some applications, like state machines, the vector index is used as a priority, e.g., to define which transition is firing if several transitions become active at the same time instant. It is then not sufficient to only provide a mechanism to always connect to the last index. Instead, some mechanism to select an index conveniently should be provided.

4. *If a connection line is deleted to an inside connector with* connectorSizing *and case 2 does not apply then, then the (*connectorSizing*) parameter is decremented by one and all connections with index above the deleted connection index are also decremented by one.* Example:*Assume there are 4 connections:*

   ```
     Step step1(nIn=4);
   equation
     connect(a1, step1.inPorts[1]);
   ```

```
  connect(a2, step1.inPorts[2]);
  connect(a3, step1.inPorts[3]);
  connect(a4, step1.inPorts[4]);
```

and the connection from `a2` *to* `step1.inPorts[2]` *is deleted. This results in*

```
  Step step1(nIn=3);
equation
  connect(a1, step1.inPorts[1]);
  connect(a3, step1.inPorts[2]);
  connect(a4, step1.inPorts[3]);
```

These rules also apply if the connectors and/or components are defined in superclass.

Example: *Assume that* `step1` *is defined in superclass* `MyCompositeStep` *with 3 connections, and a new connection is performed in a derived class. The result is:*

```
  extends MyCompositeStep(step1(nIn=4)); // new modifier nIn=4
equation
  connect(..., step1.inPorts[4]);  // new connect-equation
```

]

18.9 Versions

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.

18.9.1 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.

18.9.2 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 = Versions, [to=VERSION-NUMBER,] Convert))`
  where *Versions* is `VERSION-NUMBER | {VERSION-NUMBER, VERSION-NUMBER, ...}` and `Convert` is `script="..." | change={conversionRule(), ..., conversionRule()}`
  Defines that user models using the `VERSION-NUMBER` or any of the given `VERSION-NUMBER` can be upgraded to the given `VERSION-NUMBER` (if the to-tag is missing this is the `CURRENT-VERSION-NUMBER`)

of the current class by applying the given conversion rules. The script consists of an unordered sequence of `conversionRule();` (and optionally Modelica comments). The `conversionRule` functions are defined in section 18.9.2.1.

[*The to-tag is added for clarity and optionally allows a tool to convert in multiple steps.*]

- `uses(IDENT (version = VERSION-NUMBER [, versionBuild=INTEGER] [, dateModified=STRING ] ) )`
  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 = "3.1",
    conversion(
      noneFromVersion = "3.1 Beta 1",
      noneFromVersion = "3.1 Beta 2",
      from(version = {"2.1", "2.2", "2.2.1"},
           script = "convertTo3.mos"),
      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 = "3.1 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.*]

18.9.2.1 Conversion Rules

There are a number of functions: `convertClass`, `convertClassIf`, `convertElement`, `convertModifiers`, `convertMessage` defined as follows. The calls of these functions do not directly convert, instead they define conversion rules as below. It is recommended, but not required, to terminate each such function call with a semi-colon. The order between the function calls does not matter, instead the longer paths (in terms of number of hierarchical names) are used first as indicated below, and it is an error if there are any ambiguities.

The conversion should generate correct Modelica models using the new version of the library corresponding to the old version.

[*Whenever possible tools should preserve the original style of the model, e.g., use of imports. Conversions should be applied in all places where named element are used in code, including Modelica URIs (for example, in* `Documentation` *annotations).*]

These functions can be called with literal strings or array of strings and vectorize according to section 12.4.6.

All of these convert-functions only use inheritance among user models, and not in the library that is used for the conversion – thus conversions of base classes will require multiple conversion calls; this ensures that the conversion is independent of the new library structure. The name of the class used as argument to `convertElement` and `convertModifiers` is similarly the old name of the class, i.e., the name before it is possibly converted by `convertClass`.

[*Specifying conversions using the old name of a class allows the conversion to be done without access to the old version of the library (by suitable modifications of the lookup). Another alternative is to use the old version of the library during the conversion.*]

convertClass("OldClass", "NewClass") Convert class `OldClass` to `NewClass`.

Match longer path first, so if converting both `A` to `C` and `A.B` to `D` then `A.F` is converted to `C.F` and `A.B.E` to `D.E`. This is considered before `convertMessage` for the same `OldClass`.

[*Example: Consider the following as part of a conversion script:*

```
convertClass("Modelica.SIunits", "Modelica.Units.SI");
convertClass("Modelica.SIunits.Icons", "Modelica.Units.Icons");
```

This ensures that for example `Modelica.SIunits.Length` *is converted to* `Modelica.Units.SI.Length` *and* `Modelica.SIunits.Icons` *is converted to* `Modelica.Units.Icons`.]

convertClassIf("OldClass", "oldElement", "whenValue", "NewClass") Convert class `OldClass` to `NewClass` if the literal modifier for `oldElement` has the value `whenValue`, and also remove the modifier for `oldElement`.

These are considered before `convertClass` and `convertMessage` for the same `OldClass`.

The old element should be of a `Boolean`, `Integer`, `String`, or enumeration type and the match is based on the literal value of the modifier. For string elements the value argument to `convertClassIf` shall be up-quoted, e.g., `"\"My String\""`, and for enumeration literals only the enumeration literal part of the old value matters, e.g., `red` for `"Colors.red"`.

convertElement("OldClass", "OldName", "NewName") In `OldClass`, convert element `OldName` to `NewName`. Both `OldName` and `NewName` normally refer to components, but they may also refer to class-parameters, or hierarchical names. For hierarchical names, the longest match is used first.

For replaceable classes in packages (and replaceable classes in other classes) `convertElement` shall be used if the class is renamed within the package (or class), whereas `convertClass` shall only be used if the class is placed outside of the package (or class).

[*The latter case indicates a problem with overuse of replaceable classes in the previous design of the library.*]

[*Example: Consider the following as part of a conversion script:*

```
convertElement({"Modelica.Mechanics.MultiBody.World",
                "Modelica.Mechanics.MultiBody.World.gravityAcceleration"},
                "mue", "mu");
```

This implies that

```
Modelica.Mechanics.MultiBody.World world(mue=2);
function f=Modelica.Mechanics.MultiBody.World.gravityAcceleration(mue=4);
```

is converted to:

```
Modelica.Mechanics.MultiBody.World world(mu=2);
function f=Modelica.Mechanics.MultiBody.World.gravityAcceleration(mu=4);
```

]

convertModifiers

```
convertModifiers("OldClass",
  {"OldModifier1=default1", "OldModifier2=default2", ...},
  {"NewModifier1=...%OldModifier2%...", "NewModifier2=...", ...}
  [, simplify=true]);
```

Normal case; if any modifier among `OldModifier` exist then replace all of them with the list of `NewModifiers`. The ...`%OldModifier2%`... indicate an expression that may involve the values of the old modifiers (tools are responsible for adding parentheses if needed). The lists of old and new modifiers can have different lengths. The defaults (if present) are used if there are multiple `OldModifier` and not all are set in the component instance. The defaults are optional if there is at most one `OldModifier` element, and should otherwise be provided.

If `simplify` is specified and true then perform obvious simplifications to clean up the new modifier; otherwise leave as is.

[*Note:* `simplify` *is primarily intended for converting enumerations and emulated enumerations that naturally lead to large nested* `if`*-expressions. The simplifications may also simplify parts of the original expression.*]

If the modifiers contain literal string values they must be quoted.

Behaviour in unusual cases:

- if `NewModifier` list is empty then the modifier is just removed
- If `OldModifer` list is empty it is added for all uses of the class
- If `OldModifier`*i* is `cardinality(a) = 0` the conversion will only be applied for a component comp if there are no inside connections to `comp.a`. This can be combined with other modifiers that are handled in the usual way.
- If `OldModifier`*i* is `cardinality(a) = 1` the conversion will only be applied for a component comp if there are any inside connections to `comp.a`.

The converted modifiers and existing modifiers are merged such that the existing modifiers take precedence over the result of `convertModifiers`. A diagnostic is recommended if this merging removes some modifiers unless those modifiers are identical or it is the special case of an empty `OldModifier` list.

[*This can be used to handle the case where the default value was changed.*]

Converting modifiers with cardinality is used to remove the deprecated operator `cardinality` from model libraries, and replace tests on cardinality in models by parameters explicitly enabling the different cases. The case where the old class is used as a base class, and there exist outside connections to `a`, and there is `convertModifiers` involving the cardinality of `a` is not handled.

[*Having a parameter for explicitly enabling the different cases means that instead of model* `A` *internally testing if its connector* `B` *is connected, there will be a parameter for enabling connector* `B`*, and the conversion ensures that each component of model* `A` *will have this parameter set accordingly.*

In case a parameter is simply renamed it is preferable to use `convertElement`*, since that also handles, e.g., binding equations using the parameter.*]

[*Example: The conversion*

```
convertClass("Modelica.Thermal.FluidHeatFlow.Components.IsolatedPipe",
             "Modelica.Thermal.FluidHeatFlow.Components.Pipe");
convertModifiers({"Modelica.Thermal.FluidHeatFlow.Components.IsolatedPipe"},
                 fill("", 0), {"useHeatPort=false"});

convertClass("Modelica.StateGraph.Temporary.NumericValue",
             "Modelica.Blocks.Interaction.Show.RealValue");
convertModifiers("Modelica.StateGraph.Temporary.NumericValue",
                 {"Value"}, {"number=%Value%"});
convertModifiers("Modelica.StateGraph.Temporary.NumericValue",
                 {"hideConnector"}, {"use_numberPort=not %hideConnector%"});
```

```
convertModifiers("Modelica.Blocks.Math.LinearDependency",
   {"y0=0", "k1=0", "k2=0"}, {"y0=%y0%", "k1=%y0%*%k1%", "k2=%y0%*%k2%"},
   true);

convertClass("My.Library.BadPackage",
             "My.Library.Package");
convertElement("My.Library.BadPackage.PartialBase",
               "bad", "correct");
convertElement("My.Library.BadPackage.ActualClass",
               "bad", "correct");
```

converts

```
Modelica.Thermal.FluidHeatFlow.Components.IsolatedPipe pipe1;
Modelica.StateGraph.Temporary.NumericValue tempValue(
  Value = 10, hideConnector = true);
Modelica.Blocks.Math.LinearDependency linearDep(y0 = 2, k2 = 1);
model A
  import My.Library;
  extends Library.BadPackage.ActualClass;
end A;
model B
  extends A;
  Boolean b = bad;
end B;
```

to

```
Modelica.Thermal.FluidHeatFlow.Components.Pipe pipe1(useHeatPort = false);
Modelica.Blocks.Interaction.Show.RealValue(
  number = 10, use_numberPort = not true);
Modelica.Blocks.Math.LinearDependency linearDep(y0 = 2, k1 = 0, k2 = 2);
model A
  import My.Library;
  extends Library.Package.ActualClass;
end A;
model B
  extends A;
  Boolean b = correct;
end B;
```

The `convertElement` *call for* `ActualClass` *is needed to avoid relying on base classes in the original library where* `ActualClass` *inherits from* `PartialBase`*. However, the inheritance among the models to convert (in this case* `B` *inherits from* `A`*) should be handled. Note that conversion works regardless of the import of* `My.Library`*.*]

convertMessage("OldClass", "Failed Message") For any use of `OldClass` (or element of `OldClass`) report that conversion could not be applied with the given message.

[*This may be useful if there is no possibility to convert a specific class. An alternative is to construct* `ObsoleteLibraryA` *for problematic cases, which may be more work but allows users to directly run the models after the conversion and later convert them.*]

convertMessage("OldClass", "Failed Message", "oldElement") For any use of `oldElement` in `OldClass` report that conversion could not be applied with the given message.

[*This is useful if there is no possibility to convert a specific parameter (or other element), especially if it rarely modified. If the parameter had no impact on the model it can be removed using* `convertModifiers`*, see section 18.9.2.1.*]

18.9.3 Versions in the 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` with the file package.mo directly inside it
  Example: Modelica/package.mo
- The directory `IDENT " " VERSION-NUMBER` with the file package.mo directly inside it
  Example: Modelica 2.1/package.mo

This allows a tool to access multiple versions of the same package.

18.9.4 Version Date and Build Information

Besides version information, a top-level class can have additionally the following top-level annotations to specify associated information to the version number:

```
/*literal*/ constant String versionDate
   "UTC date of first version build (in format: YYYY-MM-DD)";
/*literal*/ constant Integer versionBuild
   "Larger number is a more recent maintenance update";
/*literal*/ constant String dateModified
   "UTC date and time of the latest change to the package
    in the following format (with one space between date
    and time): YYYY-MM-DD hh:mm:ssZ";
/*literal*/ constant String revisionId
   "Revision identifier of the version management system used
    to manage this library. It marks the latest submitted
    change to any file belonging to the package";
```

[*Example:*

```
package Modelica
  ...
  annotation(
    version = "3.0.1",
    versionDate = "2008-04-10",
    versionBuild = 4,
    dateModified = "2009-02-15 16:33:14Z",
    revisionId = "$Id:: package.mo 2566 2009-05-26 13:25:54Z #$");
end Modelica;

model M1
  annotation(
    uses(Modelica(version = "3.0.1"))); // Common case
end M1

model M2
  annotation(
    uses(Modelica(version = "3.0.1", versionBuild = 4)));
end M2
```

]

The meanings of these annotations are:

- `version` is the version number of the released library, see section 18.9.2.
- `versionDate` is the date in UTC format (according to ISO 8601) when the library was released. This string is updated by the library author to correspond with the version number.
- `versionBuild` is the optional build number of the library. When a new version is released `versionBuild` should be omitted or `versionBuild = 1`. There might be bug fixes to the library that do not justify a new library version. Such maintenance changes are called a *build* release

of the library. For every new maintenance change, the `versionBuild` number is increased. A `versionBuild` number A that is higher than `versionBuild` number B, is a newer release of the library. There are no conversions between the same versions with different build numbers.

Two releases of a library with the same `version` but different `versionBuild` are in general assumed to be compatible. In special cases, the `uses`-clause of a model may specify `versionBuild` and/or `dateModified`. In such a case the tool is expected to give a warning if there is a mismatch between library and model.

- `dateModified` is the UTC date and time (according to ISO 8601) of the last modification of the package.

  [*The intention is that a Modelica tool updates this annotation whenever the package or part of it was modified and is saved on persistent storage (like file or database system).*]

- `revisionId` is a tool specific revision identifier possibly generated by a source code management system (e.g., Subversion or CVS). This information exactly identifies the library source code in the source code management system.

The `versionBuild` and `dateModified` annotations can also be specified in the `uses` annotation (together with the version number).

[*It is recommended that tools do not automatically store* `versionBuild` *and* `dateModified` *in the* `uses` *annotation.*]

18.10 Access Control to Protect Intellectual Property

This section presents annotations to define the protection and the licensing of packages. The goal is to unify basic mechanisms to control the access to a package in order to protect the intellectual property contained in it. This information is used to encrypt a package and bind it optionally to a particular target machine, and/or restrict the usage for a particular period of time.

[*Protecting the intellectual property of a Modelica package is considerably more difficult than protecting code from a programming language. The reason is that a Modelica tool needs the model equations in order that it can process the equations symbolically, as needed for acausal modeling. Furthermore, if a Modelica tool generates C-code of the processed equations, this code is then potentially available for inspection by the user. Finally, the Modelica tool vendors have to be trusted, that they do not have a backdoor in their tools to store the (internally) decrypted classes in human readable format. The only way to protect against such misuse is legally binding warranties of the tool vendors.*

The intent of this section is to enable a library vendor to maintain one source version of their Modelica library that can be encrypted and used with several different Modelica tools, using different encryption formats.]

The following definitions relate to access control.

Definition 18.1. ***Protection.*** Define what parts of a class are visible. □

Definition 18.2. ***Obfuscation.*** Changing a Modelica class or generated code so that it is difficult to inspect by a user (e.g., by automatically renaming variables to non-meaningful names). □

Definition 18.3. ***Encryption.*** Encoding of a model or a package in a form so that the modeler cannot inspect any content of a class without an appropriate key. An encrypted package that has the `Protection` annotation is read-only; the way to modify it is to generate a new encrypted version. □

Definition 18.4. ***Licensing.*** Restrict the use of an encrypted package for particular users for a specified period of time. □

In this section annotations are defined for protection and licensing. Obfuscation and encryption are not standardized.

Protection and licensing are both defined inside the `Protection` annotation:

```
annotation(Protection(...));
```

18.10.1 Protection of Classes

A class may have the following annotations to define what parts of a class are visible, and only the parts explicitly listed as visible below can be accessed (if a class is encrypted and no `Protection` annotation is defined, the access annotation has the default value `Access.documentation`):

```
type Access =
  enumeration(hide, icon, documentation, diagram,
              nonPackageText, nonPackageDuplicate,
              packageText, packageDuplicate);
annotation(Protection(access = Access.documentation));
```

The items of the `Access` enumeration have the following meanings:

1. `Access.hide`
   Do not show the class anywhere (it is not possible to inspect any part of the class).
2. `Access.icon`
   The class can be instantiated and public parameter, constant, input, output variables as well as public connectors can be accessed, as well as the `Icon` annotation, as defined in section 18.7 (the declared information of these elements can be shown). Additionally, the class name and its description text can be accessed.
3. `Access.documentation`
   Same as `Access.icon` and additionally the `Documentation` annotation (as defined in section 18.3) can be accessed. HTML-generation in the `Documentation` annotation is normally performed before encryption, but the generated HTML is intended to be used with the encrypted package. Thus the HTML-generation should use the same access as the encrypted version – even before encryption.
4. `Access.diagram`
   Same as `Access.documentation` and additionally, the `Diagram` annotation, and all components and `connect`-equations that have a graphical annotation can be accessed.
5. `Access.nonPackageText`
   Same as `Access.diagram` and additionally if it is not a package: the whole class definition can be accessed (but that text cannot be copied, i.e., you can see but not copy the source code).
6. `Access.nonPackageDuplicate`
   Same as `Access.nonPackageText` and additionally if it is not a package: the class, or part of the class, can be copied.
7. `Access.packageText`
   Same as `Access.diagram` (note: *not* including all rights of `Access.nonPackageDuplicate`) and additionally the whole class definition can be accessed (but that text cannot be copied, i.e., you can see but not copy the source code).
8. `Access.packageDuplicate`
   Same as `Access.packageText` and additionally the class, or part of the class, can be copied.

The `access` annotation holds for the respective class and all classes that are hierarchically on a lower level, unless overridden by a `Protection` annotation with `access`. Overriding `access=Access.hide` or `access=Access.packageDuplicate` has no effect.

[*Example: If the annotation is given on the top level of a package and at no other class in this package, then the* `access` *annotation holds for all classes in this package.*]

Classes should not use other classes in ways that contradict this protection. Tools must ensure that protected contents are not shown, even if classes do not meet this requirement.

[*Example: For instance a class with* `Access.hide` *should not be used in the diagram layer of a class with* `Access.diagram`*, and there should not be hyperlinks to classes with* `Access.icon` *(from classes with visible documentation).*

Consider the following invalid use of a class with `Access.hide`*:*

```
package P
  block MySecret
```

```
    RealOutput y = time;
    annotation(Protection(access = Access.hide));
  end MySecret;
  model M
    MySecret mySecret
     annotation(Placement(
       transformation(origin = {30, 30}, extent = {{-10, -10}, {10, 10}})));
    Integrator integrator
     annotation(Placement(
       transformation(origin = {70, 30}, extent = {{-10, -10}, {10, 10}})));
  equation
    connect(mySecret.y, integrator.u)
      annotation(Line(origin = {49.5, 30}, points = {{-8.5, 0}, {8.5, -0}}));
    annotation(Protection(access = Access.diagram));
  end M;

  model M2
    // The class MySecret is a simpler Modelica.Blocks.Sources.ContinuousClock
    MySecret mySecret annotation(Placement(
       transformation(origin = {30, 30}, extent = {{-10, -10}, {10, 10}})));
    annotation(Protection(access = Access.packageDuplicate));
  end M2;
end P;
```

In order to not reveal the existence of the class `P.MySecret` *in* `P.M`*, a tool may choose to show the diagram of* `P.M` *with both* `mySecret` *and all connections to it removed. (The tool could also choose to not show the diagram of* `P.M` *at all, or even reject to load the package* `P` *altogether.) As long as the invalid use of* `P.MySecret` *occurs within the same top level package as where the class is defined (here,* `P`*), a tool is allowed to silently ignore the use for purposes of model translation. When simulating* `P.M`*, the tool must not store* `mySecret.y`*.*

It is not specified whether a tool hides the entire text of `P.M2`*, hides just the declaration, or shows the entire text of the* `P.M2`*. In order to support development of valid protected packages, it is of course OK and expected that a tool will report the invalid use of* `P.MySecret` *in* `P.M` *and* `P.M2` *(revealing its existence in a diagnostic) during development of the package.*]

[*Example: With the same package* `P` *as in the previous example, consider the following invalid use outside of* `P`*:*

```
model My
  // There exist a class P.MySecret
  P.MySecret a
    annotation(Placement(
       transformation(origin = {30, 30}, extent = {{-10, -10}, {10, 10}})));
end My;
```

Regardless of the protection of `My`*, a tool must act as if* `P.MySecret` *did not exist. For example, translation of* `My` *must fail with a standard error message about reference to the non-existing class* `P.MySecret`*. Further, just like when being misused inside the package* `P`*, the tool must not reveal that it knows about the icon of* `P.MySecret`*. With such precautions taken, showing the text or diagram of* `My` *is permitted as it doesn't reveal the actual existence of* `P.MySecret`*.*]

[*It is currently not standardized which result variables are accessible for plotting. It seems natural to not introduce new flags for this, but reuse the* `Access.XXX` *definition. For instance:*

- *For* `Access.icon` *only the variables can be stored in a result file that can also be inspected in the class.*
- *For* `Access.nonPackageText` *all public and protected variables can be stored in a result file, because all variables can be inspected in the class.*

```
package CommercialFluid // Access icon, documentation, diagram
  package Examples // Access icon, documentation, diagram
    model PipeExample // Access everything, can be copied
    end PipeExample;
```

```
    package Circuits // Access icon, documentation, diagram
      model ClosedCircuit // Access everything, can be copied
      end ClosedCircuit;
    end Circuits;

    model SecretExample // No access
      annotation(Protection(access=Access.hide));
    end SecretExample;
    annotation(Protection(access=Access.nonPackageDuplicate));
  end Examples;

  package Pipe // Access icon
    model StraightPipe // Access icon
    end StraightPipe;
    annotation(Protection(access=Access.icon));
  end Pipe;

  package Vessels // Access icon, documentation, diagram
    model Tank // Access icon, documentation, diagram, text
    end Tank;
  end Vessels;
  annotation(Protection(access=Access.nonPackageText));
end CommercialFluid;
```

]

18.10.2 Licensing

In this section annotations within the **Protection** annotation are defined to restrict the usage of the encrypted package:

```
record Protection
  ...
  /*literal*/ constant String features[:] = fill("", 0) "Required license
    features";
  record License
    /*literal*/ constant String libraryKey;
    /*literal*/ constant String licenseFile = "" "Optional, default mapping if
    empty";
  end License;
end Protection;
```

The **License** annotation has only an effect on the top of an encrypted class and is then valid for the whole class hierarchy. (Usually the licensed class is a package.) The **libraryKey** is a secret string from the library vendor and is the protection mechanism so that a user cannot generate his/her own authorization file since the **libraryKey** is unknown to him/her.

The **features** annotation defines the required license options. If the features vector has more than one element, then at least a license feature according to one of the elements must be present. As with the other annotations, the **features** annotation holds for the respective class and for all classes that are hierarchically on a lower level, unless further restricted by a corresponding annotation. If no license according to the **features** annotation is provided in the authorization file, the corresponding classes are not visible and cannot be used, not even internally in the package.

[*Example:*

```
// Requires license feature "LicenseOption"
annotation(Protection(features={"LicenseOption"}));

// Requires license features "LicenseOption1" or "LicenseOption2"
annotation(Protection(features={"LicenseOption1", "LicenseOption2"}));
```

```
// Requires license features ("LicenseOption1" and "LicenseOption2") or "
    LicenseOption3"
annotation(Protection(features={"LicenseOption1 LicenseOption2", "
    LicenseOption3"}));
```

]

In order that the protected class can be used either a tool specific license manager, or a license file (called `licenseFile`) must be present. The license file is standardized. It is a Modelica package without classes that has a `Protection` annotation of the following form which specifies a sequence of target records, which makes it natural to define start/end dates for different sets of targets individually:

```
record Authorization
  /*literal*/ constant String licensor = "" "Optional string to show
    information about the licensor";
  /*literal*/ constant String libraryKey "Matching the key in the class. Must
    be encrypted and not visible";
  /*literal*/ constant License license[:] "Definition of the license options
    and of the access rights";
end Authorization;

record License
  String licensee = "" "Optional string to show information about the licensee"
    ;
  String id[:] "Unique machine identifications, e.g., MAC addresses";
  String features[:] = fill("", 0) "Activated library license features";
  String startDate = "" "Optional start date in UTCformat YYYY-MM-DD";
  String expirationDate = "" "Optional expiration date in UTCformat YYYY-MM-DD"
    ;
  String operations[:] = fill("", 0) "Library usage conditions";
end License;
```

The format of the strings used for `libraryKey` and `id` are not specified (they are vendor specific). The `libraryKey` is a secret of the library developer. The `operations` define the usage conditions and the following are default names:

- `"ExportBinary"` Binary code generated from the Modelica code of the library can be included in binaries produced by a simulation tool.
- `"ExportSource"` Source code generated from the Modelica code of the library can be included in sources produced by a simulation tool.

Additional tool-specific names can also be used. To protect the `libraryKey` and the target definitions, the authorization file must be encrypted and must never show the `libraryKey`.

[*All other information, especially licensor and license should be visible, in order that the user can get information about the license. It is useful to include the name of the tool in the authorization file name with which it was encrypted. Note, it is not useful to store this information in the annotation, because only the tool that encrypted the* `Authorization` *package can also decrypt it.*]

[*Example: (Before encryption:)*

```
// File MyLibrary\package.mo
package MyLibrary
  annotation(Protection(License(libraryKey="15783-A39323-498222-444ckk4ll",
  licenseFile="MyLibraryAuthorization_Tool.mo_lic"), ...));
end MyLibrary;

// File MyLibrary\MyLibraryAuthorization_Tool.mo\
// (authorization file before encryption)
package MyLibraryAuthorization_Tool
  annotation(Authorization(
  libraryKey="15783-A39323-498222-444ckk4ll",
  licensor ="Organization A\nRoad, Country",
 license={
```

```
  License(licensee="Organization B, Mr.X",
    id ={"lic:1269"}), // tool license number
  License(licensee="Organization C, Mr. Y",
    id ={"lic:511"}, expirationDate="2010-06-30",
   operations={"ExportBinary"}),
  License(licensee="Organization D, Mr. Z",
    id ={"mac:0019d2c9bfe7"}) // MAC address
  }));
end MyLibraryAuthorization_Tool;
```

]

18.11 Functions

See section 12.7 *Derivatives and Inverses of Functions*, section 12.8 *Function Inlining and Event Generation*, and section 12.9.4 *Annotations for External Functions*.

18.12 Choices for Modifications and Redeclarations

See section 7.3.4 *Annotations for Redeclaration and Modification*.

Chapter 19

Unit Expressions

Unless otherwise stated, the syntax and semantics of unit expressions in Modelica (for example, section 4.9.1 or section 18.3.2.1) conform with the international standards *International System of Units (SI)* by BIPM superseding parts of 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, these standards do not define a formal syntax for unit expressions. There are recommendations and Modelica exploits them.

Note that this document uses the American spelling *meter*, whereas the SI specification from BIPM uses the British spelling *metre*.

Examples for the syntax of unit expressions used in Modelica: `"N.m"`, `"kg.m/s2"`, `"kg.m.s-2"`, `"1/rad"`, `"mm/s"`.

19.1 The Syntax of Unit Expressions

The Modelica unit string syntax allows neither comments nor white-space, and a unit string shall match the *unit-expression* rule:

```
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 SI standard does not define any precedence between multiplications and divisions. The SI standard does not allow multiple units to the right of the division-symbol (/) since the result is ambiguous; either the divisor shall be enclosed in parentheses, or negative exponents used instead of division, for example, `"J/(kg.K)"` may be written as `"J.kg-1.K-1"`.

```
unit-factors :
   unit-factor [ "." unit-factors ]
```

The SI standard specifies that a multiplication operator symbol is written as space or as a dot. The SI standard requires that this *dot* is a bit above the base line: '·', which is not part of ASCII. The ISO standard also prefers '·', but Modelica supports the ISO alternative '.', which is an ordinary *dot* on the base line.

For example, Modelica does not support `"Nm"` for newton-meter, but requires it to be written as `"N.m"`.

```
unit-factor :
  unit-operand [ unit-exponent ]
```

```
unit-exponent :
   [ "+" | "-" ] ( UNSIGNED-INTEGER | "(" UNSIGNED-INTEGER "/"
    UNSIGNED-INTEGER ")" )
```

The SI standard uses super-script for the exponentation, and does thus not define any operator symbol for exponentiation. A *unit-factor* consists of a *unit-operand* possibly suffixed by a possibly signed integer or rational number, which is interpreted as an exponent. There must be no spacing between the *unit-operand* and a possible *unit-exponent*. It is recommended to use the simplest representation of exponents, meaning that the explicit + sign should be avoided, that leading zeros should be avoided, that rational exponents are reduced to not have common factors in the numerator and denominator, that rational exponents with denominator 1 should be avoided in favor of plain integer exponents, that the exponent 1 is omitted, and that entire factors with exponent 0 are omitted.

```
unit-operand :
   unit-symbol | unit-prefix unit-symbol

unit-prefix :
   "Y" | "Z" | "E" | "P" | "T" | "G" | "M" | "k" | "h" | "da"
   | "d" | "c" | "m" | "u" | "n" | "p" | "f" | "a" | "z" | "y"

unit-symbol :
   unit-char { unit-char }

unit-char :
   NON-DIGIT
```

It is required that basic and derived units of the SI system are recognized, but tools are allowed to additionally support user-defined unit symbols. The required unit symbols do not make use of Greek letters, but a unit such as Ω is spelled out as `"Ohm"`. Similarly degree is spelled out as `"deg"`, both on its own (for angles) and as part of `"degC"`, `"degF"` and `"degRk"` for temperatures (Celsius, Fahrenheit and Rankine).

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 values of the prefixes are according to the ISO standard. The letter `u` is used as a symbol for the prefix *micro*.

[*A tool may present* `"Ohm"` *as* Ω *and the prefix* `"u"` *as* μ*. Exponents such as* `"m2"` *may be presented as* m^2*. Degrees may be presented as* °*, both for* `"deg"` *on its own (for angles) and for temperatures – e.g.,* `"degC"` *can be presented as* °*C. Note that BIPM have specific recommendations for formatting using these symbols.*]

[*Example: The unit expression* `"m"` *means meter and not milli (*10^{-3}*), since prefixes cannot be used in isolation. For millimeter use* `"mm"` *and for square meter,* m^2*, write* `"m2"`*.*

The expression `"mm2"` *means* $(10^{-3}m)^2 = 10^{-6}m^2$*. 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 20

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`. For an overview of the current version see `https://github.com/modelica/ModelicaStandardLibrary`. This is a free library that can be used without essential restrictions, e.g., in commercial Modelica simulation environments. The Modelica Standard Library is tool-neutral, and relies on a small library, ModelicaServices, that each conformant tool must implement to handle tool-specific couplings, e.g., for animation. Furthermore, other people and organizations are developing free and commercial Modelica libraries. For information about these libraries and for downloading the free libraries see `https://modelica.org/libraries/`.

Appendix A

Modelica Concrete Syntax

A.1 Lexical conventions

The following syntactic metasymbols are used (extended BNF):

Syntax	*Description*
`[ ... ]`	Optional
`{ ... }`	Repeat zero or more times
`... | ...`	Alternatives
"text"	The *text* is treated as a single token (no white-space between any characters)

The following lexical units are defined:

```
IDENT = NON-DIGIT { DIGIT | NON-DIGIT } | Q-IDENT
Q-IDENT = "'" { Q-CHAR | S-ESCAPE } "'"
NON-DIGIT = "_" | letters "a" ... "z" | letters "A" ... "Z"
DIGIT = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
Q-CHAR = NON-DIGIT | DIGIT | "!" | "#" | "$" | "%" | "&" | "(" | ")"
   | "*" | "+" | "," | "-" | "." | "/" | ":" | ";" | "<" | ">" | "="
   | "?" | "@" | "[" | "]" | "^" | "{" | "}" | "|" | "~" | " " | """
S-ESCAPE = "\'" | "\"" | "\?" | "\\"
   | "\a" | "\b" | "\f" | "\n" | "\r" | "\t" | "\v"
STRING = """ { S-CHAR | S-ESCAPE } """
S-CHAR = see below
UNSIGNED-INTEGER = DIGIT { DIGIT }
UNSIGNED-REAL =
   UNSIGNED-INTEGER  "." [ UNSIGNED-INTEGER ]
   | UNSIGNED_INTEGER [ "." [ UNSIGNED_INTEGER ] ]
     ( "e" | "E" ) [ "+" | "-" ] UNSIGNED-INTEGER
   | "."  UNSIGNED-INTEGER [ ( "e" | "E" ) [ "+" | "-" ] UNSIGNED-INTEGER ]
```

S-CHAR is any member of the Unicode character set (`https://unicode.org`; see section 13.4 for storing as UTF-8 on files) except double-quote '"', and backslash '\'.

For identifiers the redundant escapes ('`\?`' and '`\"`') are the same as the corresponding non-escaped variants ('`?`' and '`"`'). The single quotes are part of an identifier. For example, the identifiers `'x'` and `x` are different.

Note:

- White-space and comments can be used between separate lexical units and/or symbols, and also separates them. Each lexical unit will consume the maximum number of characters from the input stream. White-space and comments cannot be used inside other lexical units, except for *STRING* and *Q-IDENT* where they are treated as part of the *STRING* or *Q-IDENT* lexical unit.
- Concatenation of string literals requires a binary expression. For example, `"a" + "b"` evaluates to `"ab"`. There is no support for the C/C++ style of concatenating adjacent string literal tokens (for

example, "a" "b" becoming "ab").

- Modelica uses the same comment syntax as C++ and Java (i.e., // signals the start of a line comment and /* ... */ is a multi-line comment); comments may contain any Unicode character. Modelica also has structured comments in the form of annotations and string comments.
- In the grammar, keywords of the Modelica language are highlighted with color, for example, equation.
- Productions use hyphen as separator both in the grammar and in the text. (Previously the grammar used underscore.)

[*Within a description-string the optional tags* <HTML> *and* </HTML> *or* <html> *and* </html> *define the start and end of content that is HTML encoded.*]

A.2 Grammar

A.2.1 Stored Definition – Within

```
stored-definition :
   [ within [ name ] ";" ]
   { [ final ] class-definition ";" }
```

A.2.2 Class Definition

```
class-definition :
   [ encapsulated ] class-prefixes class-specifier

class-prefixes :
   [ partial ]
   ( class
     | model
     | [ operator ] record
     | block
     | [ expandable ] connector
     | type
     | package
     | [ pure | impure ] [ operator ] function
     | operator
   )

class-specifier :
   long-class-specifier | short-class-specifier | der-class-specifier

long-class-specifier :
   IDENT description-string composition end IDENT
   | extends IDENT [ class-modification ] description-string composition
     end IDENT

short-class-specifier :
   IDENT "=" base-prefix type-specifier [ array-subscripts ]
   [ class-modification ] description
   | IDENT "=" enumeration "(" ( [ enum-list ] | ":" ) ")" description

der-class-specifier :
   IDENT "=" der "(" type-specifier "," IDENT { "," IDENT } ")" description

base-prefix :
   [ input | output ]

enum-list :
   enumeration-literal { "," enumeration-literal }
```

```
enumeration-literal :
   IDENT description

composition :
   element-list
   { public element-list
     | protected element-list
     | equation-section
     | algorithm-section
   }
   [ external [ language-specification ]
     [ external-function-call ] [ annotation-clause ] ";"
   ]
   [ annotation-clause ";" ]

language-specification :
   STRING

external-function-call :
   [ component-reference "=" ]
   IDENT "(" [ expression-list ] ")"

element-list :
   { element ";" }

element :
   import-clause
   | extends-clause
   | [ redeclare ]
     [ final ]
     [ inner ] [ outer ]
     ( class-definition
       | component-clause
       | replaceable ( class-definition | component-clause )
         [ constraining-clause description ]
     )

import-clause :
   import
   ( IDENT "=" name
     | name [ ".*" | "." ( "*" | "{" import-list "}" ) ]
   )
   description

import-list :
   IDENT { "," IDENT }
```

A.2.3 Extends

```
extends-clause :
   extends type-specifier [ class-or-inheritance-modification ] [
    annotation-clause ]

constraining-clause :
   constrainedby type-specifier [ class-modification ]

class-or-inheritance-modification :
   "(" [ argument-or-inheritance-modification-list ] ")"

argument-or-inheritance-modification-list :
    ( argument | inheritance-modification ) { "," ( argument |
    inheritance-modification ) }
```

```
inheritance-modification :
    break ( connect-equation | IDENT )
```

A.2.4 Component Clause

```
component-clause :
   type-prefix type-specifier [ array-subscripts ] component-list

type-prefix :
   [ flow | stream ]
   [ discrete | parameter | constant ]
   [ input | output ]

component-list :
   component-declaration { "," component-declaration }

component-declaration :
   declaration [ condition-attribute ] description

condition-attribute :
   if expression

declaration :
   IDENT [ array-subscripts ] [ modification ]
```

A.2.5 Modification

```
modification :
   class-modification [ "=" modification-expression ]
   | "=" modification-expression
   | ":=" modification-expression

modification-expression :
   expression
   | break

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 :
   name [ modification ] description-string

element-redeclaration :
   redeclare [ each ] [ final ]
   ( short-class-definition | component-clause1 | element-replaceable )

element-replaceable :
   replaceable ( short-class-definition | component-clause1 )
   [ constraining-clause ]

component-clause1 :
   type-prefix type-specifier component-declaration1
```

```
component-declaration1 :
   declaration description

short-class-definition :
   class-prefixes short-class-specifier
```

A.2.6 Equations

```
equation-section :
   [ initial ] equation { equation ";" }

algorithm-section :
   [ initial ] algorithm { statement ";" }

equation :
   ( simple-expression "=" expression
     | if-equation
     | for-equation
     | connect-equation
     | when-equation
     | component-reference function-call-args
   )
   description

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
   )
   description

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 ";" }
   { elsewhen expression then
     { statement ";" }
   }
   end when

connect-equation :
   connect "(" component-reference "," component-reference ")"
```

A.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 [ relational-operator arithmetic-expression ]

relational-operator :
   "<" | "<=" | ">" | ">=" | "==" | "<>"

arithmetic-expression :
```

```
   [ add-operator ] term { add-operator term }

add-operator :
   "+" | "-" | ".+" | ".-"

term :
   factor { mul-operator factor }

mul-operator :
   "*" | "/" | ".*" | "./"

factor :
   primary [ ("^" | ".^") primary ]

primary :
   UNSIGNED-NUMBER
   | STRING
   | false
   | true
   | ( component-reference | der | initial | pure ) function-call-args
   | component-reference
   | "(" output-expression-list ")" [ array-subscripts ]
   | "[" expression-list { ";" expression-list } "]"
   | "{" array-arguments "}"
   | end

UNSIGNED-NUMBER :
   UNSIGNED-INTEGER | UNSIGNED-REAL

type-specifier :
   ["."] name

name :
   IDENT { "." IDENT }

component-reference :
   [ "." ] IDENT [ array-subscripts ] { "." IDENT [ array-subscripts ] }

result-reference :
   component-reference
   | der "(" component-reference [ "," UNSIGNED-INTEGER ] ")"

function-call-args :
   "(" [ function-arguments ] ")"

function-arguments :
   expression [ "," function-arguments-non-first | for for-indices ]
   | function-partial-application [ "," function-arguments-non-first ]
   | named-arguments

function-arguments-non-first :
   function-argument [ "," function-arguments-non-first ]
   | named-arguments

array-arguments :
   expression [ "," array-arguments-non-first | for for-indices ]

array-arguments-non-first :
   expression [ "," array-arguments-non-first ]

named-arguments: named-argument [ "," named-arguments ]

named-argument: IDENT "=" function-argument
```

```
function-argument :
   function-partial-application | expression

function-partial-application :
   function type-specifier "(" [ named-arguments ] ")"

output-expression-list :
   [ expression ] { "," [ expression ] }

expression-list :
   expression { "," expression }

array-subscripts :
   "[" subscript { "," subscript } "]"

subscript :
   ":" | expression

description :
   description-string [ annotation-clause ]

description-string :
   [ STRING { "+" STRING } ]

annotation-clause :
   annotation class-modification
```

Appendix B

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 ($v := [p; t; \dot{x}; x; y; z; m; \texttt{pre}(z); \texttt{pre}(m)]$):

$$0 = f_{\mathrm{x}}(v, c) \tag{B.1a}$$

$$z = \begin{cases} f_{\mathrm{z}}(v, c) & \text{at events} \\ \texttt{pre}(z) & \text{otherwise} \end{cases} \tag{B.1b}$$

$$m := f_{\mathrm{m}}(v, c) \tag{B.1c}$$

$$c := f_{\mathrm{c}}(relation(v)) \tag{B.1d}$$

and where

- p: Modelica variables declared as `parameter` or `constant`, i.e., variables without any time-dependency.
- t: Modelica variable `time`, the independent (real) variable.
- $x(t)$: Modelica variables of type `Real`, appearing differentiated.
- $y(t)$: Continuous-time modelica variables of type `Real` that do not appear differentiated (= algebraic variables).
- $z(t_{\mathrm{e}})$: Discrete-time modelica variables of type `Real`. These variables change their value only at event instants t_{e}. `pre(z)` are the values of z immediately before the current event occurred.
- $m(t_{\mathrm{e}})$: Modelica variables of discrete-valued types (`Boolean`, `Integer`, etc) which are unknown. These variables change their value only at event instants t_{e}. `pre(m)` are the values of m immediately before the current event occurred.

  [*For equations in* **`when`***-clauses with discrete-valued variables on the left-hand side, the form* (B.1c) *relies upon the conceptual rewriting of equations described in section 8.3.5.1.*]

- $c(t_e)$: The conditions of all **if**-expressions generated including **when**-clauses after conversion, see section 8.3.5).
- *relation*(v): A relation containing variables v_i, e.g., $v_1 > v_2$, $v_3 \geq 0$.

For simplicity, the special cases of `noEvent` and `reinit` are not contained in the equations above and are not discussed below.

The key difference between the two groups of discrete-time variables z and and m here is how they are determined. The interpretation of the solved form of (B.1c) is that given values for everything else, there is a closed-form solution for m in the form of a sequence of assignments to each of the variables of m in turn – there must be no cyclic dependencies between the equations used to solve for m. Further, each of the original model equations behind (B.1c) must be given in almost solved form:

- Non-`Integer` equations at most requiring flipping sides of the equation to obtain the used assignment form.
- For `Integer` equations the solved variable must appear uniquely as a term (without any multiplicative factor) on either side of the equation, at most requiring addition or subtraction of other terms in the equation to obtain the used assignment form.

The interpretation of the non-solved form of (B.1b) at events, on the other hand, is that at events, the discrete-time `Real` variables z are solved together with the continuous-time variables using (B.1a) and (B.1b).

[*Example: The following model demonstrates that equation* (B.1b) *does not imply that all discrete-time* `Real` *variables are given by equations in solved form, as also the discrete-time* `Real` *variables are included in* z*:*

```
model M
  discrete Real x(start = 1.0, fixed = true);
equation
  when sample(1.0, 1.0) then
    x = 3 * pre(x) - x^2; // Valid equation for discrete-time Real variable x.
  end when;
end M;
```

Another way that a discrete-time `Real` *variable can end up becoming determined by a nonlinear equation is through coupling with other variables.*

```
model M
  discrete Real x(start = 1.0, fixed = true);
  discrete Real y(start = 0.0, fixed = true);
equation
  when sample(1.0, 1.0) then
    y = x ^ 2 + 2 * exp(-time);
    x = 3 * pre(x) - y; // OK, forming nonlinear equation system with y.
  end when;
end M;
```

]

[*Example: The following model is illegal since there is no equation in solved form that can be used in* (B.1c) *to solve for the discrete-valued variable* `y`*:*

```
model M
  Boolean x;
  Boolean y;
equation
  x = time >= 1.0;
  not y = x; /* Equation in solved form, but not with respect to y. */
end M;
```

]

The generated set of equations is used for simulation and other analysis activities. Simulation proceeds as follows. First, initialization takes place, during which initial values for the states x are found, section 8.6.

Given those initial values the equations are simulated forward in time; this is the transient analysis. 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*. After initialization, simulation proceeds with transient analysis in the following way:

1. The DAE (B.1a) is solved by a numerical integration method. In this phase the conditions c of the **if**- and **when**-clauses, as well as the discrete-time variables z and m are kept constant. Therefore, (B.1a) is a continuous function of continuous variables and the most basic requirement of numerical integrators is fulfilled.
2. During integration, all relations from (B.1d) 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 determining the time instant of the next event in advance.
3. At an event instant, (B.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 at phase 1.

Note, that both the values of the conditions c as well as the values of z and m (all discrete-time `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-time variables z and m, as well as of new initial values for the states x, are determined. The change of discrete-time 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 (B.1).

At an event instant, including the initial event, the model equations are reinitialized according to the following iteration procedure:

```
known  variables: x, t, p
unkown variables: dx/dt, y, z, m, pre(z), pre(m), c

// pre(z) = value of z before event occured
// pre(m) = value of m before event occured
loop
  solve (1) for the unknowns, with pre(z) and pre(m) fixed
  if z == pre(z) and m == pre(m) then break
  pre(z) := z
  pre(m) := m
end loop
```

Clocked variables are handled similarly as z and m (depending on type), but using `previous` instead of `pre` and only solved in the first event iteration.

Solving (B.1) for the unknowns is non-trivial, because this set of equations contains not only `Real`, but also discrete-valued unknowns. Usually, in a first step these equations are sorted and in many cases the discrete-valued unknowns m can be just computed by a forward evaluation sequence. In some cases, there remain systems of equations involving m due to cyclic dependencies with y and z (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 (B.1) contains a huge number of sparse equations. Therefore, direct simulation of (B.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 (B.1) into 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 (B.1a), after the index of (B.1a) has been reduced by the Pantelides algorithm: During continuous integration, the integrator provides x and t. Then, (B.1a) is a linear or nonlinear system of equations to compute the algebraic variables y and the state derivatives $\frac{\mathrm{d}x}{\mathrm{d}t}$ and the model returns $\frac{\mathrm{d}x}{\mathrm{d}t}$ to the integrator by solving these systems of equations. Often, (B.1a) 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 C

Derivation of Stream Equations

This appendix contains a derivation of the equation for stream connectors from chapter 15.

C.1 Mixing Enthalpy

Consider a connection set with n connectors, and denote the mass flow rates `m_flow` by $\tilde{m}$. The mixing enthalpy is defined by the mass balance (the general mass-balance for a component has $\dot{m} = \sum \tilde{m}$ which simplifies for the mixing enthalpy where $m = 0$ and thus $\dot{m} = 0$)

$$0 = \sum_{j=1}^{n} \tilde{m}_j$$

and similarly the energy balance

$$0 = \sum_{j=1}^{n} \tilde{H}_j$$

with

$$\tilde{H}_j = \tilde{m}_j \begin{cases} h_{\text{mix}} & \text{if } \tilde{m}_j > 0 \\ h_{\text{outflow},j} & \text{if } \tilde{m}_j <= 0 \end{cases}$$

Herein, mass flow rates are positive when entering models (exiting the connection set). The specific enthalpy represents the specific enthalpy inside the component, close to the connector, for the case of outflow. Expressed with variables used in the balance equations we arrive at:

$$h_{\text{outflow},j} = \begin{cases} \frac{\tilde{H}_j}{\tilde{m}_j} & \text{if } \tilde{m}_j < 0 \\ \text{arbitrary} & \text{if } \tilde{m}_j \geq 0 \end{cases}$$

While these equations are suitable for device-oriented modeling, the straightforward usage of this definition leads to models with discontinuous residual equations, which violates the prerequisites of several solvers for nonlinear equation systems. This is the reason why the actual mixing enthalpy is not modelled directly in the model equations. The stream connectors provide a suitable alternative.

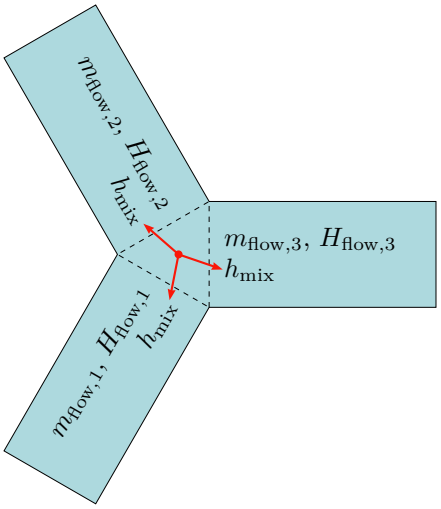


Figure C.1: *Exemplary connection set with three connected components and a common mixing enthalpy.*

C.2 Rationale for inStream

For simplicity, the derivation of `inStream` is shown at hand of 3 model components that are connected together. The case for N connections follows correspondingly.

The energy and mass balance equations for the connection set for 3 components are (see above):

$$\begin{aligned} 0 =& \tilde{m}_1 \cdot \begin{cases} h_{\text{mix}} & \text{if } \tilde{m}_1 > 0 \\ h_{\text{outflow},1} & \text{if } \tilde{m}_1 \leq 0 \end{cases} \\ &+ \tilde{m}_2 \cdot \begin{cases} h_{\text{mix}} & \text{if } \tilde{m}_2 > 0 \\ h_{\text{outflow},2} & \text{if } \tilde{m}_2 \leq 0 \end{cases} \\ &+ \tilde{m}_3 \cdot \begin{cases} h_{\text{mix}} & \text{if } \tilde{m}_3 > 0 \\ h_{\text{outflow},3} & \text{if } \tilde{m}_3 \leq \end{cases} \end{aligned} \tag{C.1a}$$

$$0 = \tilde{m}_1 + \tilde{m}_2 + \tilde{m}_3 \tag{C.1b}$$

The balance equations are implemented using a max operator in place of the piecewise expressions, taking care of the different flow directions:

$$\begin{aligned} 0 =& \max(\tilde{m}_1, 0) h_{\text{mix}} - \max(-\tilde{m}_1, 0) h_{\text{outflow},1} \\ &+ \max(\tilde{m}_2, 0) h_{\text{mix}} - \max(-\tilde{m}_2, 0) h_{\text{outflow},2} \\ &+ \max(\tilde{m}_3, 0) h_{\text{mix}} - \max(-\tilde{m}_3, 0) h_{\text{outflow},3} \end{aligned} \tag{C.2a}$$

$$\begin{aligned} 0 =& \max(\tilde{m}_1, 0) - \max(-\tilde{m}_1, 0) \\ &+ \max(\tilde{m}_2, 0) - \max(-\tilde{m}_2, 0) \\ &+ \max(\tilde{m}_3, 0) - \max(-\tilde{m}_3, 0) \end{aligned} \tag{C.2b}$$

Equation (C.2a) is solved for h_{mix}

$$h_{\text{mix}} = \frac{\max(-\tilde{m}_1, 0) h_{\text{outflow},1} + \max(-\tilde{m}_2, 0) h_{\text{outflow},2} + \max(-\tilde{m}_3, 0) h_{\text{outflow},3}}{\max(\tilde{m}_1, 0) + \max(\tilde{m}_2, 0) + \max(\tilde{m}_3, 0)}$$

Using (C.2b), the denominator can be changed to:

$$h_{\text{mix}} = \frac{\max(-\tilde{m}_1, 0) h_{\text{outflow},1} + \max(-\tilde{m}_2, 0) h_{\text{outflow},2} + \max(-\tilde{m}_3, 0) h_{\text{outflow},3}}{\max(-\tilde{m}_1, 0) + \max(-\tilde{m}_2, 0) + \max(-\tilde{m}_3, 0)}$$

Above it was shown that an equation of this type does not yield properly formulated model equations. In the streams concept we therefore decide to split the energy balance, which consists of different branches depending on the mass flow direction. Consequently, separate energy balances are the result; each valid for specific flow directions.

In a model governing equations have to establish the specific enthalpy of fluid leaving the model based on the specific enthalpy of fluid flowing into it. Whenever the mixing enthalpy is *used* in a model it is therefore the mixing enthalpy under the assumption of fluid flowing into said model.

We establish this quantity using a dedicated operator $\texttt{inStream}(h_{\text{outflow},i}) = h_{\text{mix}}$ assuming that $\tilde{m}_i \geq 0$. This leads to three different incarnations of (n in the general case). This is illustrated in the figure below. For the present example of three components in a connection set, this means the following.

$$\texttt{inStream}(h_{\text{outflow},1}) = \frac{\max(-\tilde{m}_2, 0)h_{\text{outflow},2} + \max(-\tilde{m}_3, 0)h_{\text{outflow},3}}{\max(-\tilde{m}_2, 0) + \max(-\tilde{m}_3, 0)}$$

$$\texttt{inStream}(h_{\text{outflow},2}) = \frac{\max(-\tilde{m}_1, 0)h_{\text{outflow},1} + \max(-\tilde{m}_3, 0)h_{\text{outflow},3}}{\max(-\tilde{m}_1, 0) + \max(-\tilde{m}_3, 0)}$$

$$\texttt{inStream}(h_{\text{outflow},3}) = \frac{\max(-\tilde{m}_1, 0)h_{\text{outflow},1} + \max(-\tilde{m}_2, 0)h_{\text{outflow},2}}{\max(-\tilde{m}_1, 0) + \max(-\tilde{m}_2, 0)}$$

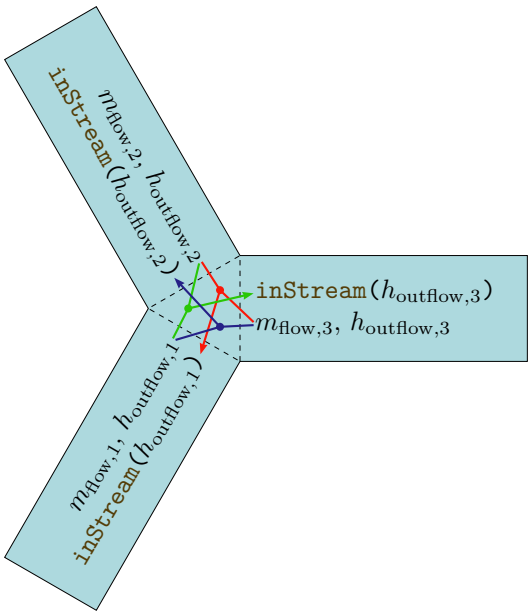


Figure C.2: *Exemplary connection set with three connected components.*

In the general case of a connection set with n components, similar considerations lead to the following.

$$\texttt{inStream}(h_{\text{outflow},i}) = \frac{\sum_{j=1,\ldots,n;j\neq i} \max(-\tilde{m}_j, 0)h_{\text{outflow},j}}{\sum_{j=1,\ldots,n;j\neq i} \max(-\tilde{m}_j, 0)}$$

C.3 Special Cases Covered by inStream Definition

C.3.1 Unconnected Stream Connector – 1 Stream Connector

For this case, the return value of `inStream` is arbitrary. Therefore, it is set to the outflow value.

C.3.2 One to One Connections – Connection of 2 Stream Connectors

$$\texttt{inStream}(h_{\text{outflow},1}) = \frac{\max(-\tilde{m}_2, 0)h_{\text{outflow},2}}{\max(-\tilde{m}_2, 0)} = h_{\text{outflow},2}$$

$$\texttt{inStream}(h_{\text{outflow},2}) = \frac{\max(-\tilde{m}_1, 0) h_{\text{outflow},1}}{\max(-\tilde{m}_1, 0)} = h_{\text{outflow},1}$$

In this case, `inStream` is continuous (contrary to h_{mix}) and does not depend on flow rates. The latter result means that this transformation may remove nonlinear systems of equations, which requires that either simplifications of the form $a * b/a = b$ must be provided, or that this case is treated directly.

C.3.3 Zero Mass Flow Rate – Connection of 3 Stream Connectors

The case where $N = 3$ and $\tilde{m}_3 = 0$ occurs when a one-port sensor (like a temperature sensor) is connected to two connected components. For the sensor, the **min** attribute of the mass flow rate should be set to zero (no fluid exiting the component via this connector). This simplification (and similar ones) can also be used if a tool determines that a mass flow rate is zero or non-negative. It is also possible to generalize this to the case where more than one sensor is connected. The suggested implementation results in the following equations, and as indicated the last formula can be simplified further by using $\tilde{m}_3 = 0$:

$$\begin{aligned}
\texttt{inStream}(h_{\text{outflow},1}) &= h_{\text{outflow},2} \\
\texttt{inStream}(h_{\text{outflow},2}) &= h_{\text{outflow},1} \\
\texttt{inStream}(h_{\text{outflow},3}) &= \frac{\max(-\tilde{m}_1, 0) h_{\text{outflow},1} + \max(-\tilde{m}_2, 0) h_{\text{outflow},2}}{\max(-\tilde{m}_1, 0) + \max(-\tilde{m}_2, 0)} \\
&= \begin{cases} h_{\text{outflow},2} & \text{if } \tilde{m}_1 \geq 0 \\ h_{\text{outflow},1} & \text{if } \tilde{m}_1 < 0 \text{ and } \tilde{m}_3 = 0 \end{cases}
\end{aligned}$$

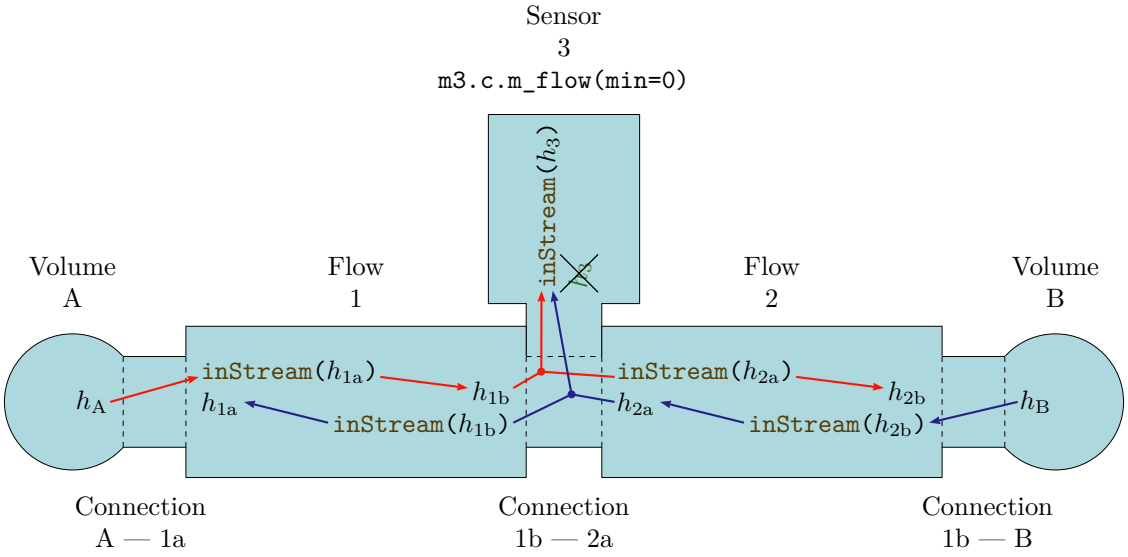


Figure C.3: *Example series connection of multiple models with stream connectors.*

For the two components with finite mass flow rates (not the sensor), the properties discussed for two connected components still hold. The connection set equations reflect that the sensor does not any influence by discarding the flow rate of the latter. In several cases a non-linear equation system is removed by this transformation. However, `inStream` results in a discontinuous equation for the sensor, which is consistent with modeling the convective phenomena only. The discontinuous equation is uncritical, if the sensor variable is not used in a feedback loop with direct feedthrough, since the discontinuous equation is then not part of an algebraic loop. Otherwise, it is advisable to regularize or filter the sensor signal.

C.3.4 Ideal Splitting Junction for Uni-Directional Flow - Connection of 3 Stream Connectors where Two Mass Flow Rates are Positive

If uni-directional flow is present and an ideal splitter is modelled, the required flow direction should be defined in the connector instance with the **min** attribute (the **max** attribute could be also defined, however

it does not lead to simplifications):

```
model m2
  Fluidport c(m_flow(min=0));
  ...
end m2;
```

Consider the case of and all other mass flow rates positive (with the min attribute set accordingly). Connecting **m1.c** with **m2.c** and **m3.c**, such that

```
m2.c.m_flow.min = 0; // max(-m2.c.m_flow,0) = 0
m3.c.m_flow.min = 0; // max(-m3.c.m_flow,0) = 0
```

results in the following equation:

$$\texttt{inStream}(h_{\text{outflow},1}) = \frac{\max(-\tilde{m}_2, 0)h_{\text{outflow},2} + \max(-\tilde{m}_3, 0)h_{\text{outflow},3}}{\max(-\tilde{m}_2, 0) + \max(-\tilde{m}_3, 0)} = \frac{0}{0}$$

`inStream` cannot be evaluated for a connector, on which the mass flow rate has to be negative by definition. The reason is that the value is arbitrary, which is why it is defined as follows.

$$\texttt{inStream}(h_{\text{outflow},1}) := h_{\text{outflow},1}$$

For the remaining connectors, `inStream` reduces to a simple result.

$$\texttt{inStream}(h_{\text{outflow},2}) = \frac{\max(-\tilde{m}_1, 0)h_{\text{outflow},1} + \max(-\tilde{m}_3, 0)h_{\text{outflow},3}}{\max(-\tilde{m}_1, 0) + \max(-\tilde{m}_3, 0)} = h_{\text{outflow},1}$$

$$\texttt{inStream}(h_{\text{outflow},3}) = \frac{\max(-\tilde{m}_1, 0)h_{\text{outflow},1} + \max(-\tilde{m}_2, 0)h_{\text{outflow},2}}{\max(-\tilde{m}_1, 0) + \max(-\tilde{m}_2, 0)} = h_{\text{outflow},1}$$

Again, the previous non-linear algebraic system of equations is removed. This means that utilizing the information about uni-directional flow is very important.

To summarize, if all mass flow rates are zero, the balance equations for stream variables (C.1) and for flows (C.2) are identically fulfilled. In such a case, any value of h_{mix} fulfills (C.1), i.e., a unique mathematical solution does not exist. This specification only requires that a solution fulfills the balance equations. Additionally, a recommendation is given to compute all unknowns in a unique way, by providing an explicit formula for `inStream`. Due to the definition, that only flows where the corresponding **min** attribute is neither zero nor positive enter this formula, a meaningful physcial result is always obtained, even in case of zero mass flow rate. As a side effect, non-linear equation systems are automatically removed in special cases, like sensors or uni-directional flow, without any symbolic transformations (no equation must be analyzed; only the **min** attributes of the corresponding flow variables).

Appendix D

Modelica Revision History

This appendix described the history of the Modelica Language Design, and its contributors. This appendix is just present for historical reasons and is not normative. The current version, as well as all previous versions of this document, and the list of changes are available at `https://specification.modelica.org`.

The members of the Modelica Association project *Modelica Language* (MAP-Lang) contributed to the Modelica specification.

Modelica 1, the first version of Modelica, was released in September 1997, and had the language specification as a short appendix to the rationale.

Bibliography

Benveniste, Albert, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone (2003). "The Synchronous Languages Twelve Years Later". In: *Proceedings of the IEEE* 91.1. URL: https://doi.org/10.1109/JPROC.2002.805826 (cit. on p. 237).

Bürger, Christoff (Mar. 2019). "Modelica language extensions for practical non-monotonic modelling: on the need for *selective* model extension". In: *Proceedings of the 13th International Modelica Conference*. Regensburg, Germany, pp. 277–288. URL: https://doi.org/10.3384/ecp19157277 (cit. on p. 106).

Colaço, Jean-Louis and Marc Pouzet (Oct. 2003). "Clocks as First Class Abstract Types". In: *Third International Workshop on Embedded Software, EMSOFT 2003*. Philadelphia, Pennsylvania, USA. URL: http://www.di.ens.fr/~pouzet/lucid-synchrone/papers/emsoft03.ps.gz (cit. on p. 237).

Drepper, Ulrich, Jim Meyering, François Pinard, and Bruno Haible (2020). *GNU gettext tools, version 0.21*. URL: https://www.gnu.org/software/gettext/manual/ (cit. on pp. 219, 220).

Elmqvist, Hilding, Martin Otter, and Francois E. Cellier (June 1995). "Inline Integration: A New Mixed Symbolic/Numeric Approach for Solving Differential-Algebraic Equation Systems". In: *Proceedings of ESM'95, European Simulation Multiconference*. Prague, Czech Republic, pp. xxiii–xxxiv. URL: https://www.semanticscholar.org/paper/Inline-Integration%3A-A-New-Mixed-Symbolic%2FNumeric-Elmqvist-Otter/b696154cbfb9c82dd4983abbd45ed639a4d5c32c (cit. on p. 255).

Forget, Julien, Frédéric Boniol, David Lesens, and Claire Pagetti (Dec. 2008). "A Multi-Periodic Synchronous Data-Flow Language". In: *11th IEEE High Assurance Systems Engineering Symposium (HASE'08)*. Nanjing, China, pp. 251–260. URL: https://doi.org/10.1109/HASE.2008.47 (cit. on p. 237).

Harel, David (1987). "Statecharts: A Visual Formalism for Complex Systems". In: *Science of Computer Programming* 8, pp. 231–274. URL: http://www.inf.ed.ac.uk/teaching/courses/seoc1/2005_2006/resources/statecharts.pdf (cit. on p. 260).

Pantelides, Constantinos C. (Mar. 1988). "The Consistent Initialization of Differential-Algebraic Systems". In: *SIAM Journal on Scientific and Statistical Computing* 9.2, pp. 213–231. URL: https://doi.org/10.1137/0909014 (cit. on p. 293).

Pouzet, Marc (2006). *Lucid Synchrone, Version 3.0, Tutorial and Reference Manual*. URL: http://www.di.ens.fr/~pouzet/lucid-synchrone/ (cit. on pp. 237, 260).

Thümmel, Michael, Gertjan Looye, Matthias Kurze, Martin Otter, and Johann Bals (Mar. 2005). "Nonlinear Inverse Models for Control". In: *Proceedings of 4th International Modelica Conference, ed. G. Schmitz*. Hamburg, Germany. URL: https://modelica.org/events/Conference2005/online_proceedings/Session3/Session3c3.pdf (cit. on pp. 238, 252).

Index

`abs`, 21
`absoluteValue`, 293
`Access`, 304
`acos`, 24
`activeState`, 261
`actualStream`, 26
algorithm
  clocked, 249
  section, 164
`algorithm`, 164
`and`, 19
`annotation`, 272
array
  constructor, 152
    with iterators, 153
  element, 145
  variable, 145
`Arrow`, 284
`asin`, 24
`assert`
  equation, 116
  statement, 171
`AssertionLevel`, 66
assignment statement
  indexed, 156
  simple, 165
`atan`, 24
`atan2`, 24
attribute, 61
`Authorization`
  license file, 307
`Axis`, 275
`AxisScale`, 275

`backSample`, 247
balanced
  globally, 56
  locally, 56
base class, 89
base-clock
  conversion-operators, 244
  partition, 240
base-partition, 240
  clocked, 250
  continuous-time, 250
binding equation, 109
  in function, 182
`Bitmap`, 290
`block`, 53
`Boolean`, 63
  reserved name, 62
`BooleanType`, 62
`BorderPattern`, 284
`break`, 169
  deselection, 106
  removing modifier, 97

`cardinality`, 25
`cat`, 154
`ceil`, 24
`change`, 34
`choices`, 104
`choicesAllMatching`, 105
class, 49
`class`, 91
class tree, 74
class type, 79, 82
`Clock`, 239
  event, 242
  inferred, 241
  interval, 242
  rational, 241
  solver, 243
clock
  inference, 249
  partition, 240
    base-, 240
    sub-, 240
  variable, 239
clocked
  algorithm, 249
  base-partition, 250
  discrete-time expression, 238
  equation, 249
  expression, 253
  state variable, 244
  variability, 253
  variable, 239
  `when`-clause, 248
clocked discrete-time, 240
`Color`, 284
comment, 11
compatible interface, 79
component, 9, 40
  declaration, 40
  expression (argument restriction), 240

reference, 12
component variability, 45
constant, 45
continuous-time, 46
discrete-time, 46
evaluable parameter, 45
non-evaluable parameter, 46
connect
equation, 126
overconstrained equation operator, 139
connection equation, 132
connection equations
connection graph, 141
stream, 231
connection graph equations, 141
Connections, 66
Connections.branch, 139
Connections.isRoot, 139
Connections.potentialRoot, 139
Connections.root, 139
Connections.rooted, 140
connector, *53*, 126
constant
expression, 36
variable, 45
constant, 45
constrainedby, 102
continuous-time
base-partition, 250
base-partition, *the*, 250
discretized, 240
expression, 38
variable, 46
conversion, 297
conversion-operators
base-clock, 244
sub-clock, 246
convertClass, 299
convertClassIf, 299
convertElement, 299
convertMessage, 301
convertModifiers, 300
CoordinateSystem, 283
cos, 24
cosh, 24
cross, 152
Curve, 276

DAE, 322
dateModified, 302
declaration assignment (deprecated), 182
declaration equation, 42, *109*
declared variability, 45
default connectable, 84
defaultComponentName, 292
defaultComponentPrefixes, 292
defaultConnectionStructurallyInconsistent, 293
definite root node, 139
delay, 25
delayed transition, 261
delimited comment, 11
der, 25
derivative, 188, 189
derivative-function, 189
derived class, 89
derived from, 89
deselection
selective model extension, 106
diagonal, 148
Diagram, 282
DiagramMap, 286
Dialog, 294
discrete, 46, 49
discrete-time
clocked, 240
expression, 36
sub-partition, 253
variable, 46
discrete-valued equation variability rule, 37
discretized
sub-partition, 252, 253
discretized continuous-time, 240
displayUnit
attribute of Real, 62
div, 23
Documentation, 273
DocumentationClass, 292
DrawingUnit, 282

each, 95
edge, 34
element, 9, 50
primitive, 132
element modification, 92
element-redeclaration, 92
Ellipse, 288
EllipseClosure, 284
else
if-equation, 111
if-expression, 20
if-statement, 169
elseif
if-equation, 111
if-expression, 20
if-statement, 169
elsewhen
when-equation, 112
when-statement, 170
empty class, 52
encapsulated, 69, 70
encryption
access control, 303
end, 157
enhancement
specialized class, 53

enumeration, 63
    conversion operator, 22
    unspecified, 66
EnumType, 62
equalityConstraint, 138
equation, 109
    clocked, 249
equation, 109
escape sequence
    string literal, 14
    text markup, 276
evaluable expression, 36
Evaluate, 278
event, 118
event clock, 242
exp, 24
expandable, 127
expandable connector, 127
experiment, 280
expression, 15
    clocked, 253
expression variability, 34
    clocked, 253
    clocked discrete-time, 238
    constant, 36
    continuous-time, 38
    discrete-time, 36
    evaluable, 36
    non-discrete-time, 38
    parameter, 36
extends, 89
Extent, 282
external, 199
ExternalObject, 211

fallback value, 62
false, 14
Figure, 274
fill, 149
FilledShape, 285
FillPattern, 284
final, 96
firstTick, 257
fixed
    attribute of Boolean, 63
    attribute of Integer, 63
    attribute of Real, 62
    attribute of String, 63
flattening, 9, 73
floor, 24
flow, 133
    in expandable connector, 129
for
    equation, 110
    reduction expression, 151
    statement, 166
function, 172
function, 53
function compatible interface, 79
function subtype, 79
function-compatibility, 85

GenerateEvents, 198
getInstanceName, 26
globally balanced, 56
GraphicItem, 283

HideResult, 280
hold, 245
homotopy, 25
hybrid DAE, 322

Icon, 282
IconMap, 286
identifier, 12
identity, 148
if
    equation, 111
    expression, 20
    statement, 169
immediate transition, 261
import, 214
import name, 69
impure, 176
in
    for-equation, 110
    reduction expression, 151
Include, 205
IncludeDirectory, 206
index, 156
inferred clock, 241
inheritance interface, 79, 82
initial, 33
initial algorithm, 120
initial equation, 120
initial value problem, 9
    seetransient analysis, 9
initialization, 9
initialization problem, 120
initialState, 261
Inline, 198
InlineAfterIndexReduction, 198
inner, 70
input, 43, 172
inside connector, 127
instance, 49
instance tree, 74
inStream, 26
Integer, 62
    conversion operator, 22
    reserved name, 62
integer, 24
IntegerType, 62
interface, 79, 81
    compatible, 79
    function compatible, 79
    inheritance, 79, 82

plug compatible, 79
interval, 257
inverse, 189, 195

keyword, 12

LateInline, 198
Library, 205
LibraryDirectory, 206
License, 306
license file, 307
licensing
access control, 303
Line, 287
connect annotation, 286
Linear (axis scale), 275
LinePattern, 284
linspace, 149
literal, 13
local equation size, 55
local number of unknowns, 55
locally balanced, 56
Log (axis scale), 276
log, 24
log10, 24
loop
for-equation, 110
for-statement, 166
while-statement, 168

matrix, 144
matrix, 148
max
attribute of Integer, 63
attribute of Real, 62
binary function, 150
of array, 150
reduction expression, 150
mayOnlyConnectOnce, 281
min
attribute of Integer, 63
attribute of Real, 62
binary function, 150
of array, 149
reduction expression, 150
missingInnerMessage, 292
mod, 23
model, 53
Modelica, 1
URI scheme, 218
ModelicaAllocateString, 211
ModelicaAllocateStringWithErrorReturn, 211
ModelicaDuplicateString, 211
ModelicaDuplicateStringWithErrorReturn, 211
ModelicaError, 210
ModelicaFormatError, 210
ModelicaFormatMessage, 210
ModelicaFormatWarning, 210
ModelicaMessage, 210
MODELICAPATH, 216
ModelicaVFormatError, 210
ModelicaVFormatMessage, 210
ModelicaVFormatWarning, 210
ModelicaWarning, 210
modification, 92
modification environment, 93
modification equation, 92, *109*
mustBeConnected, 281

name, 12
named element, 50
ndims, 147
noClock, 248
noEvent, 33
nominal
attribute of Real, 62
non-discrete-time expression, 38
non-expandable connector, 127
not, 19

obfuscation
access control, 303
obsolete, 293
ones, 149
OnMouseDownEditInteger, 291
OnMouseDownEditReal, 291
OnMouseDownEditString, 292
OnMouseDownSetBoolean, 290
OnMouseMoveXSetReal, 291
OnMouseMoveYSetReal, 291
OnMouseUpSetBoolean, 291
operator, 54
operator function, 54
operator record, 53, 222
optional spanning-tree edge, 139
or, 19
outer, 70
outerProduct, 152
output, 43, 172
outside connector, 127
overdetermined
connector, 138
record, 138
type, 138

package, 53
parameter
evaluable, 45
evaluated, 46
expression, 36
non-evaluable, 46
parameter, 46
parametric variability, *see* parameter, expression
partial, 41
partially instantiated, 75
partition
continuous-time, 250
discretized, 252

perfect matching rule, 118
piecewise-constant variable, 238
`Placement`, 285
`Plot`, 274
plug compatible interface, 79
plug-compatibility, 84
`Point`, 282
`Polygon`, 288
potential root node, 139
`pre`, 33
`preferredView`, 292
prefix, 42
`previous`, 244
primitive element, 132
primitive type, 62
`product`
  of array, 150
  reduction expression, 150
`promote`, 147
`protected`, 39
`Protection`, 303
protection
  access control, 303
`public`, 39
`pure`, 175

`quantity`
  attribute of `Boolean`, 63
  attribute of `Integer`, 63
  attribute of `Real`, 62
  attribute of `String`, 63

rational interval clock, 241
`Real`, 62
  reserved name, 62
real interval clock, 242
`RealType`, 62
`record`, 53
`Rectangle`, 288
redeclaration
  element, 92
`redeclare`, 98
reduction expression, 151
`reinit`, 34
`rem`, 23
replaceable, 100
`replaceable`, 93, 98
required spanning-tree edge, 139
rest-of-line comment, 11
restricted class, *see* specialized class
restricted subtype, 79, *see also* plug-compatibility
`return`, 173
`revisionId`, 302
root node
  definite, 139
  potential, 139

`sample`
  clocked, 245
  event-generating, 33
scalar, 144
`scalar`, 148
`Selector`, 294
`semiLinear`, 26
`shiftSample`, 247
short class definition, 51
`sign`, 21
simple type, 52
simulation, 9
simulation model, 8
`sin`, 24
single assignment rule, *see* perfect matching rule
`singleInstance`, 281
`sinh`, 24
`size`
  of all array dimensions, 148
  of single array dimension, 147
sizeless array component, 130
`skew`, 152
`Smooth`, 284
`smooth`, 33
`smoothOrder`, 188
solver clock, 243
`SourceDirectory`, 206
`spatialDistribution`, 26
specialized class, 53
`sqrt`, 21
`start`
  attribute of `Boolean`, 63
  attribute of `Integer`, 63
  attribute of `Real`, 62
  attribute of `String`, 63
state variable
  clocked, 244
`StateSelect`, 66
`stateSelect`
  attribute of `Real`, 62
stream
  connector, 230
  variable, 230
`stream`, 230
stream connection equations, 231
`String`, 63
  conversion operator, 22
  reserved name, 62
`StringType`, 62
structural parameter, 48
sub-clock
  conversion-operators, 246
  partition, 240
sub-partition, 240
  discrete-time, 253
  discretized, 253
`subSample`, 246
subscript, 156
subtype, 79
  function, 79

functions, 85
restricted, 79
sum
of array, 150
reduction expression, 150
superSample, 246
supertype, 79
symmetric, 152

tan, 24
tanh, 24
terminal, 33
terminate
equation, 117
statement, 171
TestCase, 280
Text, 289
connect annotation, 286
text markup escape sequence, 276
TextAlignment, 284
TextStyle, 284
then
if-equation, 111
if-expression, 20
if-statement, 169
when-equation, 112
when-statement, 170
ticksInState, 261
time, 20
timeInState, 261
Transformation, 285
transient analysis, 9
transition
delayed, 261
immediate, 261
transition, 261
transitively non-replaceable, 82
translation
multilingual libraries, 219
of simulation model, 9
transpose, 152
true, 14
type, 81
class, 79, 82
interface, 79
type, 53

unassignedMessage, 293
unbounded
attribute of Real, 62
unit
attribute of Real, 62
URI
Modelica, 218
uses, 298

variability
declared, *see* declared variability
expression, *see* expression variability
prefix, 43
variable, 40
vector, 144
vector, 148
vendor-specific annotation, 272
vendor-specific markup, 278
version, 297
versionBuild, 302
versionDate, 302

when
equation, 112
statement, 170
when-clause
clocked, 248
while
statement, 168
within, 217

zeros, 149