Modelica Language Specification Documentation

Release 3.3 Revision 1

Modelica Association

June 09, 2015

Contents:

CHAPTER

ONE

MODELICA CHANGE PROPOSAL: CHANGING MODELICA LANGUAGE SPECIFICATION DOCUMENTATION FORMAT

1.1 Summary

1.1.1 Revisions

Version	Date	Comments
v1	2015-06-08	Initial version

1.1.2 Copyright License

1.2 Motivation

1.2.1 Benefits of using restructured text (Sphinx)

It is a text-based representation

Equations are converted to nice vector graphics:

$$\frac{\partial x}{\partial t} = 0.0$$

Easy to get diff's for equations as they are also written in textual markup (latex).

Easier to merge changes using a textual format than a binary one. This makes collaboration much simpler, in particular when working on making sure MCP's are up-to-date.

Easy to track individual changes (each commit is textual and easy to see).

We could work on the specification directly during the design or web meetings.

Restructured text specifics

Easier to reference particular subsections or items (a static, named, link instead of an ever-changing section number). That is, how to reference the specification from for example an e-mail or forum discussion.

Automatic syntax highlighting for Modelica code snippets (no more trying to remember which words to boldface):

```
model BouncingBall
  parameter Real e=0.7 "coefficient of restitution";
  parameter Real g=9.81 "gravity acceleration";
  Real h(start=1) "height of ball";
  Real v "velocity of ball";
  Boolean flying(start=true) "true, if ball is flying";
  Boolean impact;
  Real v_new;
  Integer foo;
equation
  impact = h <= 0.0;
  foo = if impact then 1 else 2;
  der(v) = if flying then -g else 0;
  der(h) = v;

  when {h <= 0.0 and v <= 0.0,impact} then
    v_new = if edge(impact) then -e*pre(v) else 0;
    flying = v_new > 0;
    reinit(v, v_new);
  end when;
end BouncingBall;
```

Good search functionality (the JavaScript is able to search the html even when the HTML is generated locally - no server required).

Google indexes the generated HTML easily.

It is also possible to host different versions of the documentation on https://readthedocs.org.

http://sphinx-doc.org/ has good documentation on how to use restructured text.

There are many useful features available:

- Abbreviations like using :abbr: MSL (Modelica Standard Library) or as a short-hand name |MLS| MLS (Modelica Language Specification)
- Citations like **:cite:‘something‘** (working with bibtex) is available as plugins.
- Easy generation of an index, like if and when.
- Easy to theme the generated HTML.

Generation of the text into multiple formats:

- HTML (the primary format, web searchable and linkable)
- PDF (via LaTeX)
- epub (e-book readers)
- qthelp (minor benefit; tools can use the built-in help and load the spec in it)

Benefits of using git

Easier to propose changes, for example by writing an MCP in restructured text or creating a github pull request for fixing typo’s.

Possible to work on the specification during meetings even when the Wi-Fi is failing.

1.2.2 Impact

We get a version of the specification that:

- Can be easily modified.
- Supports collaboration.
- Can be converted to different formats easily.

1.2.3 Use Cases

See *Impact*.

1.3 Proposed Changes in Specification

The specification will need to be updated. The initial conversion can be done by pandoc with manual changes of the specification to make sure all formatting has been preserved, indexes updated, source code using source code blocks, etc.

1.3.1 Required changes in List of Keywords

Conversion to restructured text.

1.3.2 Required changes in Grammar

Conversion to restructured text.

1.3.3 Required changes in Specification Text

Conversion to restructured text.

1.4 Backwards Compatibility

No actual changes will be made.

1.5 Implementation Effort

No changes will be made that directly affect Modelica tools.

CHAPTER

TWO

INTRODUCTION

2.1 Overview of Modelica

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

2.2 Scope of the Specification

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

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

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

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

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

The flat hybrid DAE form consists of:

- Declarations of variables with the appropriate basic types, prefixes and attributes, such as `parameter Real v=5`.
- Equations from equation sections.
- Function invocations where an invocation is treated as a set of equations which involves all input and all result variables (number of equations = number of basic result variables).

- Algorithm sections where every section is treated as a set of equations which involves the variables occurring in the algorithm section (number of equations = number of different assigned variables).
- When-clauses where every when-clause is treated as a set of conditionally evaluated equations, also called instantaneous equations, which are functions of the variables occurring in the clause (number of equations = number of different assigned variables).

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

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

2.3 Some Definitions

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

2.3.1 Component

An element defined by the production component_clause in the Modelica grammar (basically a variable or an instance of a class).

2.3.2 Element

Class definitions, extends-clauses and component-clauses declared in a class (basically a class reference or a component in a declaration).

2.3.3 Flattening

The translation of a model described in Modelica to the corresponding model described as a hybrid DAE, involving expansion of inherited base classes, parameterization of base classes, local classes and components, and generation of connection equations from connect-equations (basically, mapping the hierarchical structure of a model into a set of differential, algebraic and discrete equations together with the corresponding variable declarations and function definitions from the model).

Todo

Reference the glossary. Create the glossary.

2.4 Notation and Grammar

The following syntactic meta symbols are used (extended BNF [1]):

[ ] optional

{ } repeat zero or more times

[1] ISO/IEC 14977 EBNF grammar.

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

See TODO: Appendix B for a full lexical specification and grammar.

CHAPTER

THREE

LEXICAL STRUCTURE

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

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

3.1 Character Set

The character set of the Modelica language is Unicode, but restricted to the Unicode characters corresponding to 7-bit ASCII characters in several places; for details see Appendix B.1.

3.2 Comments

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

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

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

```
/* Commented out - erroneous comment, invalid nesting of comments!
/* This is an interesting model */

model interesting
  // ...
end interesting;
*/
```

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

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

3.3 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.

3.3.1 Identifiers

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

```
IDENT = NONDIGIT { DIGIT | NONDIGIT } | Q-IDENT ;
Q-IDENT = "'" { Q-CHAR | S-ESCAPE } "'" ;
NONDIGIT = "_" | letters "a" to "z" | letters "A" to "Z" ;
DIGIT = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
Q-CHAR = NONDIGIT | DIGIT | "!" | "#" | "$" | "%" | "&" | "(" | ")" | "\*" | "+" |
  "," | "-" | "." | "/" | ":" | ";" | "<" | ">" | "=" | "?" | "@" | "[" | "]" |
  "^" | "{" | "}" | "|" | "~" | " " | "_" ;
S-ESCAPE = "\" ( "'" """" "?" "\" "a" "b" "f" "n" "r" "t" "v" ) ;
```

3.3.2 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 todo: Chapter 5. The meaning of package names is described in more detail in todo: Chapter 13.

3.3.3 Modelica Keywords

The following Modelica *keywords* are reserved words and may not be used as identifiers, except as listed in TODO: Appendix B.1:

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

3.4 Literal Constants

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

3.4.1 Floating Point Numbers

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

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

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

```
13., 13E0, 1.3e1, 0.13E2
```

3.4.2 Integer Literals

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

3.4.3 Boolean Literals

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

3.4.4 Strings

String literals appear between double quotes as in "between". Any character in the Modelica language character set (see appendix B.1 for allowed characters) apart from double quote (") and backslash (\), including new-line, can be *directly* included in a string without using an escape code. Certain characters in string literals can be represented using escape codes, i.e., the character is preceded by a backslash (\) within the string. Those characters are:

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

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

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

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

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

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

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

]

3.5 Operator Symbols

The predefined operator symbols are formally defined on todo: page 255 and summarized in the table of operators in Table 4.1.

CHAPTER

FOUR

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 TODO: Appendix B.

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

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

4.1 Expressions

Modelica equations, assignments and declaration equations contain expressions.

Expressions can contain basic operations, `+`, `-`, `*`, `/`, `^`, etc. with normal precedence as defined in Table 4.1 and the grammar in Section modelica-concrete-syntax. The semantics of the operations is defined for both scalar and array arguments in TODO: 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 TODO: Section 12.4.1 and for vectorized calls in TODO: Section 12.4.4. The built-in array functions are given in TODO: Section 10.1.1 and other built-in operators in Section *Built-in Intrinsic Operators with Function Syntax*.

4.2 Operator Precedence and Associativity

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

The following table presents all the expression operators in order of precedence from highest to lowest, as derived from the Modelica grammar in TODO: Appendix B. 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. Operators with the same precedence occur at the same line of the table:

Table 4.1: Operators.

Operator Group	*Operator Syntax*	*Examples*
postfix array index operator	`[]`	`arr[index]`
postfix access operator	`.`	`a.b`
postfix function call	`funcName(function-arguments)`	`sin(4.36)`
array construct/concat	`{expressions}` `[expressions]` `[expressions; expressions, ...]`	`[2,3; 7,8]` `{2,3}` `[5,6]`
exponentiation	`^`	`2^3`
multiplicative and array elementwise multiplicative	`* / .* ./`	`2*3` `2/3` `[1,2;3,4].*[2,3;5,6]`
additive and array elementwise additive	`+ - +expr -expr .+ .-`	`a+b, a-b, +a, -a` `[1,2;3,4].+[2,3;5,6]`
relational	`< <= > >= == <>`	`a<b, a<=b, a>b, ...`
unary negation	`not expr`	`not b1`
logical and	`and`	`b1 and b2`
logical or	`or`	`b1 or b2`
array range	`expr : expr` `expr : expr : expr`	`1:5` `start:step:stop`
conditional	`if expr then expr else expr`	`if b then 3 else x`
named argument	`ident = expr`	`x = 2.26`

The conditional operator may also include elseif-clauses. Equality = and assignment := are not expression operators since they are allowed only in equations and in assignment statements respectively. All binary expression operators are left associative, except exponentiation which is non-associative. The array range operator is non-associative.

[*The unary minus and plus in Modelica is slightly different than in Mathematica and in MATLAB* [1]*, 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

Non-associative exponentation and array range operator:

`x^y^z` Not legal, use parenthesis to make it clear.

`a:b:c:d:e:f:g` Not legal, and scalar arguments gives no legal interpretation.

]

4.3 Evaluation Order

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

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

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

4.3.1 Example: Guarding Expressions Against Incorrect Evaluation

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

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

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

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

]

4.4 Arithmetic Operators

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

^	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 TODO: 10.6.

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

```
arithmetic_expression =
  [ add_op ] term { add_op term } ;

add_op =
  "+" | "-" ;

term =
  factor { mul_op factor } ;

mul_op =
  "*" | "/" ;

factor =
  primary [ "^" primary ] ;
```

4.5 Equality, Relational, and Logical Operators

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

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

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

The following logical operators are defined:

`not`	negation, unary operator
`and`	logical and
`or`	logical or

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

```
logical_expression =
  logical_term { or logical_term } ;
logical_term =
  logical_factor { and logical_factor } ;
logical_factor =
  [ not ] relation ;
relation =
  arithmetic_expression [ rel_op arithmetic_expression ] ;
rel_op =
  "<" | ( "<=" ) | ">" | ( ">=" ) | ( "==" ) | ( "<>" ) ;
```

The following holds for relational operators:

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

4.6 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.

4.6.1 String Concatenation

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

4.6.2 Array Constructor Operator

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

4.6.3 Array Concatenation Operator

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

4.6.4 Array Range Operator

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

4.6.5 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 (TODO: Section 6.6) giving the type of the if-expression. If-expressions with elseif are defined by replacing elseif by else if. [*Note:* elseif *has been added for symmetry with if-clauses.*] For short-circuit evaluation see Section *Evaluation Order*.

[*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;
```

]

4.6.6 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.*]

4.6.7 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;
```

]

4.7 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 *Numeric Functions and Conversion Functions* below.
- Derivative and special purpose operators with function syntax, see Section *Derivative and Special Purpose Operators with Function Syntax* below.
- Event-related operators with function syntax, see Section *Event-Related Operators with Function Syntax* below.
- Array operators/functions, see Section TODO: 10.1.1.

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

4.7.1 Numeric Functions and Conversion Functions

The following mathematical operators and functions, also including some conversion functions, are predefined in Modelica, and are vectorizable according to Section 12.4.6, except for the String function. The functions which do not trigger events are described in the table below, whereas the event-triggering mathematical functions are described in Section *Event Triggering Mathematical Functions*.

`abs(v)`	Is expanded into `noEvent(if v >= 0 then v else -v)`. Argument v needs to be an Integer or Real expression.
`sign(v)`	Is expanded 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.
`sqrt(v)`	Returns the square root of `v` if `v>=0`, otherwise an error occurs. Argument `v` needs to be an Integer or Real expression.
`Integer(e)`	Returns the 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 TODO 4.8.5.2.

`String(b, <options>)`
`String(i, <options>)`
`String(r, significantDigits=d, <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 e` (Section TODO: 4.8.5.2). The other arguments must use named arguments. The optional <options> are:
	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. For Real expressions the output shall be according to the Modelica grammar. Integer significantDigits=6: defines the number of significant digits in the result string. [*Examples:* `"12.3456"`, `"0.0123456"`, `"12345600"`, `"1.23456E-10"`].
	The format string corresponding to options is:
	• for Reals: `(if leftJustified then "-" else "")+String(minimumLength)+"."+String(signficantDigits)+"g"`,
	• for Integers: `(if leftJustified then "-" else "")+String(minimumLength)+"d"`.
	Format string: According to ANSI-C the format string specifies one conversion specifier (excluding the leading %), may not contain length modifiers, and may not use "*" for width and/or precision. For all numeric values the format specifiers f, e, E, g, G are allowed. For integral values it is also allowed to use the d, i, o, x, X, u, and c-format specifiers (for non-integral values a tool may round, truncate or use a different format if the integer conversion characters are used). The x,X-formats (hexa-decimal) and c (character) for Integers does not lead to input that agrees with the Modelica-grammar.

Event Triggering Mathematical Functions

The built-in operators in this section trigger state events if used outside of a when-clause and outside of a clocked discrete-time partition (see Section 16.8.1). [*If this is not desired, the* noEvent *function can be applied to them. E.g.* noEvent(integer(v))]

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

Todo

This is using an alternative to using a table: headings. There are other alternatives (like definition lists). Using headers makes it easy to get permanent links to the text.

div(x,y)

`div(x,y)` returns the algebraic quotient x/y with any fractional part discarded (also known as truncation toward zero). [*Note: this is defined for / in C99; in C89 the result for negative numbers is implementation-defined, so the standard function div() must be used.*]. Result and arguments shall have type Real or Integer. If either of the arguments is Real the result is Real otherwise Integer.

mod(x,y)

`mod(x,y)` returns the 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].

rem(x,y)

`rem(x,y)` returns the 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].

ceil(x)

`ceil(x)` returns the smallest integer not less than x. Result and argument shall have type Real. [*Note, outside of a when-clause state events are triggered when the return value changes discontinuously.*].

floor(x)

`floor(x)` returns the largest integer not greater than x. Result and argument shall have type Real. [*Note, outside of a when-clause state events are triggered when the return value changes discontinuously.*].

integer(x)

`integer(x)` returns the largest integer not greater than x. The argument shall have type Real. The result has type Integer. [*Note, outside of a when-clause state events are triggered when the return value changes discontinuously.*].

Todo

This is using an alternative to using a table: definition lists.

`div(x,y)` returns the algebraic quotient x/y with any fractional part discarded (also known as truncation toward zero). [*Note: this is defined for / in C99; in C89 the result for negative numbers is implementation-defined, so the standard function div() must be used.*]. Result and arguments shall have type Real or Integer. If either of the arguments is Real the result is Real otherwise Integer.

`mod(x,y)` returns the integer modulus of x/y, i.e. 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].

`rem(x,y)` returns the 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].

`ceil(x)` returns the smallest integer not less than x. Result and argument shall have type Real. [*Note, outside of a when-clause state events are triggered when the return value changes discontinuously.*].

`floor(x)` returns the largest integer not greater than x. Result and argument shall have type Real. [*Note, outside of a when-clause state events are triggered when the return value changes discontinuously.*].

`integer(x)` returns the largest integer not greater than x. The argument shall have type Real. The result has type Integer. [*Note, outside of a when-clause state events are triggered when the return value changes discontinuously.*].

Built-in Mathematical Functions and External Built-in Functions

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

`sin(x)`	sine
`cos(x)`	cosine
`tan(x)`	tangent (x shall not be: ..., -π/2, π/2, 3π/2, ...)
`asin(x)`	inverse sine (-1 ≤ x ≤ 1)
`acos(x)`	inverse cosine (-1 ≤ x ≤ 1)
`atan(x)`	inverse tangent
`atan2(y, x)`	the `atan2(y, x)` function calculates the principal value of the arc tangent of `y/x`, using the signs of the two arguments to determine the quadrant of the result
`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`)

4.7.2 Derivative and Special Purpose Operators with Function Syntax

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

Todo

This is using a definition list for the operators. There are other alternatives (tables are not that nice). Definition lists can be referred to using links, like *der(expr)*, but the link will not be visible in the HTML. Could possibly be themed into the HTML in some way.

`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 subexpressions must be 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 TODO: 10.6.12. [*For Real parameters and constants the result is a zero scalar or array of the same size as the variable.*]

`delay(expr,delayTime, delayMax),delay(expr,delayTime)` Returns: 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 <= delayTime <= delayMax, otherwise an error occurs. If delayMax is not supplied in the argument list, delayTime need to be a parameter expression. See also Section *delay*. For non-scalar arguments the function is vectorized according to Section TODO: 10.6.12.

`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. See also Section *cardinality (deprecated)*.

`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" *[a 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, *[e.g., actual*lambda + simplified*(1-lambda), where lambda is a homotopy parameter going from 0 to 1]*.

The solution must fulfill the equations for homotopy returning `actual`. See also Section *homotopy*. For non-scalar arguments the function is vectorized according to Section TODO: 12.4.6.

`semiLinear(x, positiveSlope, negativeSlope)` Returns: `if x>=0 then positiveSlope*x else negativeSlope*x`. The result is of type Real. See Section *semiLinear* [*especially in the case when x = 0*]. For non-scalar arguments the function is vectorized according to Section TODO: 10.6.12.

`inStream(v)` The operator `inStream(v)` is only allowed on 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 more details see Section TODO: 15.2.

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

`spatialDistribution(in0, in1, x, pv, iP, iV)` The `spatialDistribution(...)` operator allows approximation of variable-speed transport of properties, see Section *spatialDistribution*.

`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, see Section *getInstanceName*.

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

delay

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

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

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(y, 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.*]

The spatialDistribution() operator allows to approximate efficiently the solution of the infinite-dimensional problem:

$$\frac{\partial z(y,t)}{\partial t} + v(t)\frac{\partial z(y,t)}{\partial y} = 0.0$$

$$z(0.0,t) = in_0(t) \text{ if } v >= 0$$

$$z(1.0,t) = in_1(t) \text{ if } v < 0$$

where *z(y, t)* is the transported quantity, *y* is the normalized spatial coordinate ($0.0 \leq y \leq 1.0$), *t* is the time, *v*(*t*)=**der**(*x*) is the normalized transport velocity and the boundary conditions are set at either *y* = 0.0 or *y* = 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, x, v are all subtypes of Real, positiveVelocity is a Boolean, initialPoints and initialValues are arrays of subtypes of Real of equal size, containing the y coordinates and the *z* values of a finite set of points describing the initial distribution of *z(y, 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:

= $z(y, t)$, for all, as long as staying inside the domain.

This allows to directly compute the solution based on interpolating the boundary conditions.

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

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

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.:

Technically relevant use cases for the use of the spatialDistribution*() operator 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.*]

cardinality (deprecated)

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

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

]

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

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

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

]

The cardinality is counted after removing conditional components. and may 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). The cardinality operator should only be used in the condition of assert and if-statements – that do not contain connect (and similar operators – see section 8.3.3).

Listing 4.1: Pseudo-code for spatialDistribution in terms of 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
  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, (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 (if m>0 then points[m]+(x-x0)>=1 else false) then
        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] .+ (x1-x0), {1} );
    elseif x<x0 then
      m:=1;
      while (if m<size(points,1) then points[m]+(x-x0)<=0 else false) then
        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] .+ (x1-x0), {1});
    end if;
    x0:=x;
  end when;
initial algorithm
  x0:=x;
  points:=initialPoints;
  values:=initialValues;
end spatialDistribution;
```

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*actual + (1-lambda)*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 lambda, 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:

$\mathbf{w} = \mathbf{f}_1(\mathbf{x})$ // has homotopy operator
$\mathbf{0} = \mathbf{f}_2(\text{der}(\mathbf{x}), \mathbf{x}, \mathbf{z}, \mathbf{w})$

Here, a non-linear equation system $\mathbf{f}_2$ is present. The homotopy operator 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 $\mathbf{f}_1$ and $\mathbf{f}_2$ 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 a homotopy operator is present and all equations up to the last BLT block that describes a non-linear algebraic equation system.

A trivial implementation of the homotopy operator 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 the homotopy operator 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.SIunits.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 SI = Modelica.SIunits;
  // ...
  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 the homotopy operator **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*der(x) + (1-lambda)*(x-x0)
```

and you can solve the two equations to give

```
x = (lambda+(lambda-1)*x0)/(2*lambda - 1)
```

which has the correct value of x0 at lambda = 0 and of 1 at lambda = 1, but unfortunately has a singularity at lambda = 0.5.

]

semiLinear

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

Rule 1: The equations

```
y = semiLinear(x, sa, s[1]);
y = semiLinear(x, s[1], s[2]);
y = semiLinear(x, s[2], s[3]);
// ...
y = semiLinear(x, s[n], sb);
// ...
```

may be replaced by

```
s[1] = if x >= 0 then sa else sb
s[2] = s[1];
s[3] = s[2];
// ...
s[n] = s[n-1];
y = semiLinear(x, sa, sb);
```

Rule 2: The equations

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

may be replaced by

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

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

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

is identical to :

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

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

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

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

]

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.*]

4.7.3 Event-Related Operators with Function Syntax

The following event-related operators with function syntax are supported. The operators noEvent, pre, edge, and change, are vectorizable according to Section 12.4.6

`initial()` Returns true during the initialization phase and false otherwise [*thereby triggering a time event at the beginning of a simulation*].

`terminal()` Returns true at the end of a successful analysis [*thereby ensuring an event at the end of successful simulation*].

`noEvent(expr)` Real elementary relations within expr are taken literally, i.e., no state or time event is triggered. See also Section *noEvent and smooth* and Section 8.5.

`smooth(p, expr)` If p>=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. See also Section *noEvent and smooth*.

`sample(start,interval)` Returns true and triggers time events at time instants `start + i*interval(i=0,1, ...)`. During continuous integration the operator returns always false. The starting time start and the sample interval interval need to be parameter expressions and need to be a subtype of Real or Integer. |

`pre(y)` Returns the "left limit" $y(t^{pre})$ of variable y(t) at a time instant t. At an event instant, $y(t^{pre})$ is the value of y after the last event iteration at time instant t (see comment below). The pre() operator can be applied if the following three conditions are fulfilled simultaneously:

1. variable y is either a subtype of a simple type or is a record component

2. y is a discrete-time expression
3. the operator is *not* applied in a function class.

[*Note: This can be applied to continuous-time variables in when-clauses, see Section :ref:'discrete-time-expressions' for the definition of discrete-time expression.*] The first value of pre(y) is determined in the initialization phase. See also Section operator-pre.

`edge(b)` Is expanded into `(b and not pre(b))` for Boolean variable `b`. The same restrictions as for the `pre()` operator apply (e.g. not to be used in function classes).

`change(v)` Is expanded into `(v<>pre(v))`. The same restrictions as for the `pre()` operator apply.

`reinit(x, expr)` In the body of a when clause, reinitializes `x` with `expr` at an event instant. `x` is a Real variable (or an array of Real variables) that is implicitly defined to have `StateSelect.always` [*so must be selected as a state, and it is an error, if this is not possible*]. expr needs to be type-compatible with x. The `reinit` operator 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 TODO: 8.3.6.

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

pre

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

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

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

noEvent and smooth

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

[*Example*:

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

]

4.8 Variability of Expressions

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

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

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

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

4.8.1 Constant Expressions

Constant expressions are:

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

Components declared as constant shall have an associated declaration equation with a constant expression, if the constant is 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.

4.8.2 Parameter Expressions

Parameter expressions are:

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

4.8.3 Discrete-Time Expressions

Discrete-time expressions are:

- Parameter expressions.
- Discrete-time variables, i.e., Integer, Boolean, String variables and enumeration variables, as well as Real variables assigned in when-clauses
- Function calls where all input arguments of the function are discrete-time expressions.

- Expressions where all the subexpressions are discrete-time expressions.
- Expressions in the body of a when-clause, initial equation, or initial algorithm.
- Unless inside noEvent: Ordered relations (>, <, >=, <=) if at least one operand is a subtype of Real (i.e. Real elementary relations, see Section *Equality, Relational, and Logical Operators*) and the functions `ceil`, `floor`, `div`, `mod`, `rem`. These will generate events if at least one subexpression is not a discrete-time expression. [*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.

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

Inside an if-expression, if-clause, 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 variables, equations between discrete-time expressions, or real elementary relations/functions that should generate events. [*This restriction is necessary in order to guarantee that there all equations for discrete variable are discrete-time expressions, and to ensure that crossing functions do not become active between events.*]

[*Example*:

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

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

]

4.8.4 Continuous-Time Expressions

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

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

I

K

W