μ-Modelica Language Specification.

Joaquín Fernández Ernesto Kofman
CIFASIS-CONICET, Rosario, Argentina
{fernandez, kofman}@cifasis-conicet.gov.ar

Abstract

This document defines the μ-Modelica language. μ-Modelica is part of the Stand–Alone QSS solver, which is an implementation of the Quantized State System (QSS) integration methods for continuous and hybrid system simulation. A difficulty imposed by the QSS methods is that it makes use of structural information of the model and this information must be given on the form of incidence matrices. To overcome this difficulty, the solver has a Modeling Front–End that allows the user to describe models in a standard way and then it automatically generates all the structure matrices required.

μ-Modelica is defined as a sub–set of the standard Modelica language. Modelica is a free high level, object-oriented language for modeling of large, complex, and heterogeneous systems. Models in Modelica are mathematically described by differential, algebraic and discrete equations.

The μ-Modelica specification is based on the Modelica Specification, version 3.3 and it contains only the necessary Modelica keywords and structures to define an ODE based hybrid model.

Contents

Chapter 1

Lexical Structure

This section describes several of the basic building blocks of μ-Modelica such as characters and lexical units including identifiers and literals. Without question, the smallest building blocks in μ-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 μ-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.

1.1 Character Set

The character set of the μ-Modelica language is Unicode, but restricted to the Unicode characters corresponding to 7-bit ASCII characters in several places.

1.2 Comments

There are two kinds of comments in μ-Modelica which are not lexical units in the language and therefore are treated as whitespace by a μ-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.

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

```
/* Commented out - erroneous comment, invalid nesting of comments!

/* This is a interesting model */
model interesting
  ...
end interesting;
*/
```

1.3 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 *μ*-Modelica grammar and are therefore not available as identifiers.

1.3.1 Identifiers

μ-Modelica identifiers, used for naming classes, variables, constants, and other items, must always start with a letter or underscore (_), followed by any number of letters, digits, or underscores (unlike Modelica which also allows the definition of quoted literal strings). Case is significant, i.e., the names **Inductor** and **inductor** are different. The following BNF-like rules define *μ*-Modelica identifiers, where curly brackets { } indicate repetition zero or more times, and vertical bar | indicates alternatives.

⟨*IDENT*⟩	=	NONDIGIT { DIGIT | NONDIGIT }
⟨*NONDIGIT*⟩	=	`"_"` | letters `"a"` to `"z"` | letters `"A"` to `"Z"`
⟨*STRING*⟩	=	`"` S-CHAR | S-ESCAPE `"`
⟨*S-CHAR*⟩	=	any member of the Unicode character set except double-quote `"""`, and backslash `\`.
⟨*S-ESCAPE*⟩	=	`"’"` | `\"` | `"?"` | `"\"` | `"\a"` | `"\b"` | `"\f"` | `"\n"` | `"\r"` | `"\t"` | `"\v"`
⟨*DIGIT*⟩	=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
⟨*UNSIGNED_INTEGER*⟩	=	DIGIT { DIGIT }
⟨*UNSIGNED_NUMBER*⟩	=	UNSIGNED_INTEGER [. [UNSIGNED_INTEGER]] [(`e` | `E`) [`+` | `-`] UNSIGNED_INTEGER]

1.3.2 Names

A name is an identifier with a certain interpretation or meaning. A name may denote an `Integer` variable, a `Real` variable, a function, etc. In *μ*-Modelica the name always have the same meaning in all the model given that all models arew flattened, i.e. there is only one scope defined where all the names are visible. The exception to this rule are `for indexes` which are only visible inside the for loop definition.

1.3.3 μ-Modelica Keywords

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

`in`	`for`	`loop`	`end`	`if`
`then`	`else`	`elseif`	`output`	`each`
`model`	`elsewhen`	`annotation`	`equation`	`when`
`reinit`	`algorithm`	`time`	`pre`	`function`
`end`	`output`	`input`	`initial`	`der`
`parameter`	`constant`	`boolean`	`discrete`	`start`
`Real`	`Integer`	`description`	`external`	

The following keywords are used by the QSS solver, they are defined as annotations in μ-Modelica and may not be used as identifiers:

`comminterval`	`sampled`	`dense`	`step`
`dqmin`	`dqrel`	`linear`	`include`
`symDiff`	`searchmethod`	`binary`	`derDelta`
`StepSize`	`QssSettings`	`minStep`	`zcHist`
`Tolerance`	`AbsTolerance`	`solver`	`StepSize`
`experiment`	`StartTime`	`StopTime`	`description`
`QSS`	`CQSS`	`LIQSS`	`QSS2`
`LIQSS2`	`QSS3`	`LIQSS3`	

1.4 Literal Constants

Literal constants are unnamed constants that have different forms depending on their type. Each of the predefined types in μ-Modelica has a way of expressing unnamed constants of the corresponding type, which is presented in the ensuing subsections.

1.4.1 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

1.4.2 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.

1.4.3 Boolean Literals

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

Chapter 2

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

Expressions can contain variables and constants, which have predefined types. The predefined built-in types of μ-Modelica are `Real`, `Integer` and `Boolean` types which are presented in more detail in Section 3.2.

2.1 Expressions

μ-Modelica equations, assignments and declaration equations contain expressions. Expressions can contain basic operations, $+, -, *, /, \hat{}$, etc. with normal precedence as defined in Section 3.2. It is also possible to define functions and call them in a normal fashion.

2.2 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. All operators are binary except the postfix operators and those shown as unary together with `expr`, the conditional operator. Operators with the same precedence occur at the same line of the table:

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

Operator Group	*Operator Syntax*	*Examples*
postfix array index operator	`[]`	`arr[index]`
postfix function call	$funcName(function-arguments)$	`sin(3.46)`
exponentiation	ˆ	`2^3`
multiplicative	$*\ /$	`2*3, 2/3`
aditive	$+\ -\ +expr\ -expr$	`a+b, a-b, +a, -a`
relational	$<\ >\ <=\ >=$	`a < b, a > b,...`
unary negation	*not expr*	`not a`
logical and	*and*	`a and b`
logical or	*or*	`a and b`
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.

2.3 Arithmetic Operators

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

ˆ	Exponentiation
$*$	Multiplication
/	Division
$+$	Addition
-	Subtraction

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

```
⟨arithmetic_expression⟩ ::= [ ⟨add_op⟩ ] ⟨term⟩ { ⟨add_op⟩ ⟨term⟩ }

⟨add_op⟩          ::= "+"
                    |  "-"

⟨term⟩            ::= ⟨factor⟩ { ⟨mul_op⟩ ⟨factor⟩ }

⟨mul_op⟩          ::= "*"
                    |  "/"

⟨factor⟩          ::= ⟨primary⟩ [ "^" ⟨primary⟩ ]
```

2.4 Equality, Relational, and Logical Operators

μ-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

A single equals sign = is never used in relational expressions, only in equations. 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.
- In relations of the form $v1 == v2$ or $v1 <> v2$, $v1$ or $v2$ shall not be used in zero–crossing functions, i.e. as expressions of `when` statements, unless used in a function.
- 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 $v1$ and $v2$ shall be `Real` varaibles and the relation is called a *Real elementary relation.*

2.4.1 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, which is treated as an input variable. The initial value of the time variable is set to the time instant at which the simulation is started.

2.4.2 Built-in Intrinsic Operators with Function Syntax

Certain built-in operators of μ-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. The following built-in intrinsic operators/functions are available:

1. Mathematical functions and conversion functions.
2. Derivative and special purpose operators with function syntax.

All operators in this section can only be called with positional arguments.

2.4.3 Numeric Functions and Conversion Functions

The following mathematical operators and functions, also including some conversion functions, are predefined in μ-Modelica.

$sqrt(v)$	Returns the square root of v if $v >= 0$, otherwise an error occurs. Argument v needs to be an Integer or Real expression.
$boolToReal(x)$	Convert the boolean argument x to 1.0 if $x = true$ or 0.0 if $x = false$

2.4.4 Built-in Mathematical Functions and External Built-in Functions

The following built-in mathematical functions are available in μ-Modelica and can be called directly.

$sin(x)$	sine
$cos(x)$	cosine
$tan(x)$	tangent
$asin(x)$	inverse sine $(-1 \leq x \leq 1)$
$acos(x)$	inverse cosine $(-1 \leq x \leq 1)$
$atan(x)$	inverse tangent
$atan2(y, x)$	the $atan2(y, x)$ function calculates the principal Modelica Language Specification 3.3 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 $(x > 0)$

2.4.5 Derivative and Special Purpose Operators with Function Syntax

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

$der(expr)$	The time derivative of *expr*. If the expression *expr* is a scalar it needs to be Real. The expression and all its subexpressions must be differentiable. [*For Real parameters and constants the result is a zero scalar.*]

Chapter 3

Predefined types and Declarations

This chapter describes μ-Modelica predefined types, variability prefixes and the restrictions impossed to varaibles declarations. All μ-Modelica model variables are of type `Real` and each varaible can belong to one of the following cathegories:

- continuous states.
- discretes states.
- algebraic variables.

3.1 Component Variability Prefixes discrete, parameter, constant

The prefixes `discrete`, `parameter` and `constant` of a component declaration are called variability prefixes and define in which situation the variable values of a component are initialized (see Section 4.4 and Section 4.5) and when they are changed in transient analysis (= solution of initial value problem of the hybrid DAE):

- A variable vc declared with the `parameter` or `constant` prefixes remains constant during transient analysis.
- A discrete-time variable vd has a vanishing time derivative (informally $der(vd) = 0$, but it is not legal to apply the $der()$ operator to discrete-time variables) and can change its values only at event instants during transient analysis (see Section 4.4).
- A continuous-time variable vn may have a non-vanishing time derivative ($der(vn) <> 0$ possible) and may also change its value discontin-

uously at any time during transient analysis (see Section 4.4). If there are any discontinuities the variable is not differentiable.

If a `Real` variable is declared with the prefix `discrete` in a simulation model it must be assigned in a when-clause by an assignment. A `Real` variable not assigned in any when-clause and without any type prefix is a continuous-time variable. 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, their values remains constant during simulation.

3.2 Predefined Types

μ-Modelica has three predefined types:

- `Real`
- `Integer`
- `Bool`

These predefined types are the only types allowed on μ-Modelica, the names are reserved such that it is illegal to declare an element with these names.

The different combinations of predefined types and types refixes allowd in μ-Modelica are described by following rules:

- The `constant` prefix can only be used with `Integer` variables.
- The `parameter` or `discrete` prefixes can only be used with `Real` variables.

Variables prefixes are not allowed with `Boolean` type.

Unlike Modelica, in μ-Modelica all discrete-time variable must be defined with the `discrete` prefix, it is an error to modify a variable inside a when-clause that is not declared with the `discrete` prefix. [*Example:*

```
model bball
  Real y(start = 10),vy(start = 0), F;
  parameter Real m = 1, b = 30, g = 9.8, k = 1e6;
  Real contact(start = 0); // Error, contact must be declared discrete.
  equation
    F = k*y+b*vy;
    der(y) = vy;
    der(vy) = -g - (contact * F)/m;
  algorithm
  when y < 0 then
    contact := 1;
  elsewhen y > 0 then
    contact := 0;
  end when;
end bball;
```

The model above represents a bouncing ball with valid Modelica code, but if the contact variable is not defined with the discrete prefix it is not valid μ-Modelica code.]

3.3 Attribute start

The attribute `start` is only allowed for `Real` types (the variable can have the `discrete` prefix), furthermore, if the variable is not declare as `discrete` is must be a state variable, i.e., the *der*() operation has to be defined for that variable. It is an error to assign initial values using the `start` attribute to an algebraic variable.

[*Example:*

```
model bball
  // Error, F is an algebraic variable.
  Real y(start = 10),vy(start = 0), F(start=10);
  parameter Real m = 1, b = 30, g = 9.8, k = 1e6;
  discrete Real contact(start = 0);
  equation
    F = k*y+b*vy;
    der(y) = vy;
    der(vy) = -g - (contact * F)/m;
  algorithm
  when y < 0 then
    contact := 1;
  elsewhen y > 0 then
    contact := 0;
  end when;
end bball;
```

]

Chapter 4

Equations

4.1 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.
- Initial equations, which are used to express equations for solving initialization problems (Section 4.5)

4.2 Equations in Equation Sections

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

⟨*simple_expression*⟩ `"="` ⟨*expression*⟩
  | ⟨*name*⟩ ⟨*function_call_args*⟩

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

4.2.1 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 μ-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. Two examples:

- $simple_expr1 = expr2;$
- $(out1, out2, out3) = function_name(inexpr1, inexpr2);$

4.2.2 For-Equations – Repetitive Equation Structures

The syntax of a for-equation is as follows:

⟨*for_equation*⟩ := `for` ⟨*for_index*⟩ `loop {` ⟨*equation*⟩ `";" } end for ";"`

Multiple iterators are not allowed in μ-Modelica, the syntax of the indexes is the following:

⟨*for_index*⟩ ::= ⟨*IDENT*⟩ `in` ⟨*expression*⟩

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

```
for IDENT in expression loop
```

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

```
for i in 1:10 loop // i takes the values 1,2,3,...,10
for r in 1.0:1.5:5.5 loop // r takes the values 1.0, 2.5, 4.0, 5.5
```

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;
```

]

4.2.3 reinit

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

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

The operator reinitializes `x` with `expr` at an event instant, `x` is a `Real` variable (or an array of `Real` variables) that must be selected as a state (resp., states) at least when the enclosing when clause becomes active, `expr` needs to be type-compatible with `x`. The `reinit` operator can for the same variable (resp. array of variables) only be applied (either as an individual variable or as part of an array of variables) either in one equation (having reinit of the same variable in when and else-when of the same variable is allowed) or one or more times in one algorithm section.

The `reinit` operator does not break the single assignment rule, because `reinit(x,expr)` evaluates `expr` to a value (values), and then performs the assignment "$x := value$" in an `algorithm` section.

[*Example for the usage of the reinit operator:*

```
der(h) = v;
der(v) = flying*-g;
when h <= 0 then
flying := 0;
reinit(v, -e*pre(v));
end when;
when h > 0 then
flying := 1;
end when;
```

]

4.2.4 assert

An equation or statement of the following form:

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

is an assertion, where condition is a `Boolean` expression, `message` is a string expression, and `level` is a built-in enumeration with a default value. It can be used in equation sections or algorithm sections. If the condition of an assertion is `true`, `message` is not evaluated and the procedure call is ignored. If the condition evaluates to `false` different actions are taken depending on the level input:

- *level = AssertionLevel.error* : The current evaluation is aborted. The simulation may continue with another evaluation [*e.g., with a* *shorter step-size, or by changing the values of iteration variables*]. If the simulation is aborted, message indicates the cause of the error. Failed assertions takes precedence over successful termination, such that if the model first triggers the end of successful analysis by reaching the stop-time, 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.*].

[*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");
```

]

4.3 Synchronous Data-flow Principle and Single Assignment Rule

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

1. All variables keep their actual values until these values are explicitly changed. Variable values can be accessed at any time instant during continuous integration and at event instants.

2. At every time instant, during continuous integration and at event instants, the active equations express relations between variables which have to be fulfilled concurrently (equations are not active if the corresponding if-branch, when-clause or block in which the equation is present is not active).

3. Computation and communication at an event instant does not take time. [*If computation or communication time has to be simulated, this property has to be explicitly modeled*].

4. The total number of equations is identical to the total "number of unknown variables" (= single assignment rule).

4.4 Events and Synchronization

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

```
equation
y = du;
algorithm
when u > uMax then
   du := uMax;
end when;

when u < uMin then
  du:= uMin;
elsewhen u >= uMin then
  du :=  u;
end when;
```

The integration is halted whenever $u - uMax$ or $u - uMin$ crosses zero. At the event instant, the correct statement is selected and the integration is restarted. Numerical integration methods of order n ($n >= 1$) 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 = discreteexpression", i.e., the event instant is known in advance and no iteration is needed to find the exact event instant.] μ-Modelica is based on the synchronous data flow principle. [*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".*

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.*]

4.5 Initialization, initial equation, and initial algorithm

Before any operation is carried out with a μ-Modelica model [*e.g., simulation or linearization*], initialization takes place to assign consistent values for all variables present in the model. During this phase, also the derivatives, *der*(..) , and the pre-variables, *pre*(..) , are interpreted as unknown algebraic variables. The initialization uses all equations and algorithms that are utilized in the intended operation [*such as simulation or linearization*].

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

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

2. Implicitly by using the attributes `start=value` in the declaration of variables:

   - For all continuous-time `Real` variables v , the equation $v = startExpression$ is added to the initialization equations, if $start = startExpression$.
   - For all discrete variables vd , the equation $vd = startExpression$ is added to the initialization equations, if $start = startExpression$.
   - For all variables declared as `constant` and `parameter`; no equation is added to the initialization equations.

If a `parameter` has a start-expression, and neither has a binding equation nor is part of a record having a binding equation, the start-expression can be used as parameter-expression although a diagnostic message is recommended when initializing the model without setting the parameter value.

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

Chapter 5

Arrays

An array can be regarded as a collection of values, all of the same type. 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. An array is allocated by declaring an array variable. Elements of an array are indexed by an `Integer` value.

5.1 Array Declarations

The following restrictions are defined for array declarations in μ-Modelica:

- Only one dimension arrays are allowed (i.e. *vectors*).
- All arrays must have `Real` type.
- The only valid form of declaration is:

  ```
  {discrete} Real x[n];
  ```

  where n is the size of the array and must be an `Integer` constant. Additionaly, the `discrete` type prefix can be used to define a discrete variable array.

Upper and lower array dimension index bounds are described in Section 5.1.1.

5.1.1 Array Dimension Lower and Upper Index Bounds

The lower and upper index bounds for a dimension of an array are indexed by `Integer` values and have a lower bound of 1 and an upper bopund being the *size* of the array.

5.2 Array Indexing

The array indexing operator name [...] is used to access array elements for retrieval of their values or for updating these values. An indexing operation is subject to upper and lower array dimension index bounds (Section 5.1.1). [*An indexing operation is assumed to take constant time, i.e., largely independent of the size of the array.*]

The indexing operator takes two operands, where the first operand is the array to be indexed and the second operand are index expressions:

`arrayname[indexexpr]`

It is also possible to use the array access operator to assign to element/elements of an array in `algorithm` and `equation` sections. If the index is an array the assignments take place in the order given by the index array. For assignments to arrays and elements of arrays, the entire right-hand side and the index on the left-hand side is evaluated before any element is assigned a new value.

Indexes in arrays inside for statements in `algorithm` or `equation` sections are restricted to expressions of the form:

$$\alpha \cdot i + \beta$$

where α and β are integer expressions and i is the iteration index.

Chapter 6

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 μ-Modelica. Statements are imperative constructs allowed in algorithm sections.

6.1 Algorithm Sections

Algorithm sections is comprised of the keyword `algorithm` followed by a sequence of statements. The formal syntax is as follows:

$\langle algorithm\_section \rangle$ ::= [`initial`] `algorithm` `{` $\langle statement \rangle$ `";"`
  | `annotation` `";"` `}`

Equation equality = or any other kind of equation (see Chapter 4) shall not be used in an algorithm section.

6.1.1 Initial Algorithm Sections

See Section 4.5 for a description of both initial algorithm sections and initial equation sections.

6.1.2 Execution of 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 non-discrete variable is initialized with its start value (i.e. the value of the start-attribute).
- A discrete variable v is initialized with `pre(v)`.

[*Initialization is performed, in order that an algorithm section cannot introduce a "memory" (except in the case of discrete states which are explicitly given), 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. Conceptually the algorithm can be viewed as $(lhs1, lhs2, ...) = someFunction(nonLhs1, nonLhs2, ...)$*, where* lhs *are the variables assigned and* $nonLhs$ *are other appearing variables. 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.*]

6.2 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*⟩
  | `return`
  | ⟨*if_statement*⟩

| ⟨*for_statement*⟩
| ⟨*when_statement*⟩)

6.2.1 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`.

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 $m <= n$ receiving variables on the left-hand side.

$$(out1, out2, out3) := function\_name(in1, in2, in3, in4);$$

It is possible to omit receiving variables from this list:

$$(out1, , out3) := function\_name(in1, in2, in3, in4);$$

[*Example: The function f called below has three results and two inputs:*

$$(a, b, c) := f(1.0, 2.0);$$
$$(x[1], x[2], x[3]) := f(3, 4);$$

]

The syntax of an assignment statement with a call to a function with multiple results is as follows:

$$"("output\_expression\_list")"" := "component\_reference(function\_call\_args)$$

[*Also see Section 4.2.1 regarding calling functions with multiple results within equations.*]

6.2.2 For-statement

The syntax of a for-statement is as follows:

⟨*for_statement*⟩ := `for` ⟨*for_index*⟩ `loop {` ⟨*statement*⟩ `";" } end for ";"`

⟨*for_index*⟩ ::= ⟨*IDENT*⟩ `in` ⟨*expression*⟩

The following is an example of a prefix of a for-statement:

```
for IDENT in expression loop
```

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

[*Example:*

```
 for i in 1:10 loop          // i takes the values 1,2,3,...,10.
 for i in 1: 1.5: 5.5 loop   // r takes the values 1.0, 2.5, 4.0, 5.5
```

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;
```

]

6.2.3 Return-Statements

Can only be used inside functions, see Section 7.1.2.

6.2.4 If-Statement

If-statements have the following syntax:

```
⟨if_statement⟩ ::= 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.

6.2.5 When-Statements

A when-statement has the following syntax:

⟨*when_statement*⟩ ::= `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 algorithmic statements within a when-statement are activated when the scalar or any one of the elements of the vector-expression becomes true.

[*Example: Algorithms are activated when x becomes* > 2*:*

```
when x > 2 then
  y1 := sin(x);
  y3 := 2*x + y1+y2;
end when;
```

This is a valid Modelica condition that is not supported in μ*-Modelica. (Completar) The statements inside the when-statement are activated when either x becomes* > 2 *or sample*$(0, 2)$ *becomes true or x becomes* < 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.]

Restrictions on When-Statements

- A when-statement shall not be used within a function.
- When-statements cannot be nested.
- When-statements may not occur inside if, and for-clauses in algorithms.

[*Example: The following nested when-statement is invalid:*

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

]

6.3 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.

Reinit Statement

See Section 4.2.3.

Assert Statement

See Section 4.2.4. A failed assert stops the execution of the current algorithm.

Chapter 7

Functions

This chapter describes the Modelica function construct.

7.1 Function Declaration

A μ-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, functions are defined using the keyword `function`. Formal parameters are specified using the `input` keyword, whereas results are denoted using the `output` keyword. [*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;
  ...
  output TypeO1 out1;
  output TypeO2 out2;
  ...
  protected
  < local variables >
  ...
  algorithm
  ...
  < statements >
  ...
  end functionname;
```

] It is not allowed to use default values with any `input` or `output` formal parameter through declaration assignments. [*Example:*

```
function functionname
  input TypeI1 in1 := default1;
  ...
  output TypeO1 out1 := default2;
  ...
  end functionname;
```

In the above example, both formal parameters definitions are wrong (note that this example is valid Modelica code).

[*All internal parts of a function are optional; i.e., the following is also a legal function:*

```
function functionname
end functionname;
```

]

7.1.1 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 >;
```

]

7.1.2 Function Return-Statement

The return-statement terminates the current function call. It can only be used in an algorithm section of a function. It has the following form:

`return;`

[*Example:*

```
function max
  input Real x;
  input Real y;
  output Real res;
  algorithm
    index := x;
    if y > x then
      res := y;
      return;
    end if;
    return;
end max;
```

]

7.2 Pure Modelica Functions

μ-Modelica functions are pure, i.e., are side-effect free with respect to the μ-Modelica state (the set of all μ-Modelica variables in a total simulation model), apart from the exceptional case specified further below. This means that:

- Pure μ-Modelica functions are mathematical functions, i.e. calls with the same input argument values always give the same results.
- A pure μ-Modelica function is side-effect free with respect to the internal μ-Modelica simulation state. Specifically, the ordering of function calls and the number of calls to a function shall not influence the simulation state.
- A μ-Modelica function which does not have the pure function properties is impure and needs to be declared as stated below.

[*Comment: 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 prefixes impure or pure
- a when-statement.

With the prefix keyword pure it is stated that a μ-Modelica function is pure even though it may call impure functions.

7.3 Function Call

Functions can be called as described in this section.

Positional or Named Input Arguments of Functions

μ-Modelica function calls has only positional arguments, such as

```
f(3.5, 5.76, 5, 8.3);
```

The formal syntax of a function call:

$\langle primary \rangle$ ::= $\langle name \rangle$ $\langle function\_call\_args \rangle$

$\langle name \rangle$ ::= $\langle IDENT \rangle$

$\langle function\_call\_args \rangle$ ::= `"("` [$\langle function\_arguments \rangle$] `")"`

$\langle function\_arguments \rangle$ ::= $\langle function\_argument \rangle$ [`","` $\langle function\_arguments \rangle$]

$\langle function\_argument \rangle$ ::= $\langle expression \rangle$

The interpretation of a function call is as follows: A list of unfilled slots is created for all formal input parameters. There shall be no remaining unfilled slots [otherwise an error occurs] and the list of filled slots is used as the argument list for the call. The type of each argument must agree with the type of the corresponding parameter, except where the standard type coercions can be used to make the types agree.

7.3.1 Output Formal Parameters of Functions

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.

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.

7.3.2 Initialization and Declaration Assignments of Components in Functions

Components in a function can be divided into three groups:

- Public components which are input formal parameters.
- Public components which are output formal parameters.
- Protected components which are local variables, parameters, or constants.

When a function is called components of a function do not have start-attributes. However, a declaration
`assignment ( := expression )`
with an expression may be present for a component. A declaration assignment for a non-input component initializes the component to this expression at the start of every function invocation (before executing the algorithm section or calling the external function). These bindings must be executed in an order where a variable is not used before its declaration assignment has been executed; it is an error if no such order exists (i.e. the binding must be acyclic). Declaration assignments can only be used for components of a function. If no declaration assignment is given for a non-input component its value at the start of the function invocation is undefined. It is a quality of implementation issue to diagnose this for non-external functions.

7.4 Built-in Functions

There are basically four groups of built-in functions in Modelica:

- Intrinsic mathematical and conversion functions.
- Derivative and special operators with function syntax.
- Event-related operators with function syntax.
- Built-in array functions.

7.5 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 μ-Modelica external function call interface provides the following:

- Support for external functions written in C.
- 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 μ-Modelica to standard libraries of the target language.
- Handling arbitrary parameter order for the external function.
- Passing arrays to and from external functions where the dimension sizes are passed as explicit integer parameters.
- Handling of external function parameters which are used both for input and output.

The format of an external function declaration is as follows.

```
function IDENT
  { component_clause ";" }
  [ protected { component_clause ";" } ]
  external [ language_specification ] [ external_function_call ]
  [annotation ] ";"
  [ annotation ";" ]
end IDENT;
```

Components in the public part of an external function declaration shall be declared either as input or output. [*This is just as for any other function. The components in the protected part allows local variables for temporary storage to be declared.*] The `language_specification` must be C̈. 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 μ-Modelica name. The only permissible kinds of expressions in the argument list are identifiers and scalar constants. The annotations are used to pass additional information to the compiler when necessary.

7.5.1 Argument type Mapping

The arguments of the external function are declared in the same order as in the μ-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 are passed as inputs.

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 *

Arrays

Unless an explicit function call is present in the external declaration, an array is passed by its address followed by n arguments of type *size_t* with the corresponding array dimension sizes, where n is the number of dimensions. [*The type size_t is a C unsigned integer type.*]

7.5.2 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.

7.6 Annotations for External Libraries and Include Files

The following annotations are useful in the context of calling external functions from Modelica:

- The `annotation(Library="libraryName")`, used by the linker to include the library file where the compiled external function is available.
- The `annotation(Library={"libraryName1","libraryName2"})`, used by the linker to include the library files where the compiled external function is available and additional libraries used to implement it. For shared libraries it is recommended to include all non-system libraries in this list.

- The `annotation(Include="includeDirective")`, used to include source files, [*e.g., header files or source files that contain the functions referenced in the external function declaration*], needed for calling the external function in the code generated by the μ-Modelica compiler.

[*Example: to show the use of external functions and of object libraries:*

Chapter 8

Packages

8.1 Motivation and Usage of Packages

Packages in μ-Modelica can only contain function definitions. Parameters, constans, variables and models 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.
- Name conflicts between definitions in different packages are eliminated since the package name is implicitly prefixed to names of definitions declared in a package.

8.2 Importing Definitions from a Package

The import-clause makes functions definitions declared in some package available for use by shorter names in a model or a package. It is the only way of referring to definitions declared in some other package for use inside an encapsulated package or function. [*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 model.
- Explicit declaration of usage dependences on other packages.

] An import-clause can occur in the following syntactic form:

```
import packagename; (single definition import)
```

Here packagename is the fully qualified name of the imported package including possible dot notation.

8.2.1 Lookup of Imported Names

This section only defines how the imported name is looked up in the import clause. Lookup of the name of an imported package, e.g. A in the clause import A;. Is defined in the following way:

- Look for the package A in the local working directory.
- Look for the package A in the package folder specified in the directory structure. This feature is implementation dependent.

8.2.2 Summary of Rules for Import Clauses

The following rules apply to import-clauses:

- Import-clauses are not inherited.
- Import-clauses are not named elements of a class or package.
- The order of import-clauses does not matter.
- One can only import from packages.
- An imported package or definition should always be referred to by its fully qualified name in the import-clause.

8.2.3 Mapping Package/Class Structures to a Hierarchical File System

Packages in μ-Modelica are represented as nonstructured entities [*e.g. a file in the file system*] where all the function definitions are located.

Mapping a Package/Class Hierarchy into a Single File (Nonstructured Entity)

A nonstructured entity [*e.g. the file A.mo*] shall contain only a stored-definition that defines a package [A] with a name matching the name of the nonstructured entity.

Chapter 9

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 9.1. The only requirement is that any tool shall save files with all annotations from this chapter and all vendor-specific annotations intact. To ensure this, annotations must be represented with constructs according to the Modelica grammar. The specification in this document defines the semantic meaning if a tool implements any of these annotations.

9.1 Vendor-Specific Annotations

A vendor may – anywhere inside an annotation – add specific, possibly undocumented, annotations which are not intended to be interpreted by other tools. Two variants of vendor-specific annotations exist; one simple and one hierarchical. Double underscore concatenated with a vendor name as initial characters of the identifier are used to identify vendor-specific annotations. [*Example:*

```
annotation (
  Icon(coordinateSystem(extent={{-100,-100}, {100,100}}),
  graphics={__NameOfVendor(Circle(center={0,0}, radius=10))})
);
```

This introduces a new graphical primitive Circle using the hierarchical variant of vendor-specific annotations.

```
annotation (
  Icon(coordinateSystem(extent={{-100,-100}, {100,100}}),
  graphics={Rectangle(extent={{-5,-5},{7,7}}, __NameOfVendor_shadow=2)})
);
```

This introduces a new attribute __NameOfVendor_shadow for the Rectangle primitive using the simple variant of vendor-specific annotations.]

9.2 Annotations for Simulation Experiments

⟨*experiment_annotation*⟩ ::= `annotation` "(" `experiment` "(" [⟨*experiment_option*⟩]
    `{ ,` ⟨*experiment_option*⟩`}`] ")"

⟨*experiment_option*⟩ ::= `StartTime` "=" ["+" | "-"] UNSIGNED_NUMBER
    | `StopTime` "=" ["+" | "-"] UNSIGNED_NUMBER
    | `Interval` "=" UNSIGNED_NUMBER
    | `Tolerance` "=" UNSIGNED_NUMBER

The experiment annotation defines the default start time (`StartTime`) in [s], the default stop time (`StopTime`) in [s], the suitable time resolution for the result grid (`Interval`) in [s], and the default relative integration tolerance (`Tolerance`) for simulation experiments to be carried out with the model or block at hand.

Chapter 10

μ-Modelica Concrete Syntax

10.1 Lexical conventions

The following syntactic meta symbols are used (extended BNF):
[ ] optional
{ } repeat zero or more times
| or
"text" The text is treated as a single token (no whitespace between any characters) The following lexical units are defined (the ones in boldface are the ones used in the grammar, the rest are just internal to the definition of other lexical units):

⟨*IDENT*⟩	::=	⟨*NONDIGIT*⟩ `{` ⟨*DIGIT*⟩ | ⟨*NONDIGIT*⟩ `}`
⟨*NONDIGIT*⟩	::=	`"_"` | letters `"a"` to `"z"` | letters `"A"` to `"Z"`
⟨*STRING*⟩	::=	`"` ⟨*S-CHAR*⟩ | ⟨*S-ESCAPE*⟩ `"`
⟨*S-CHAR*⟩	=	any member of the Unicode character set except double-quote `"""`, and backslash `\`.
⟨*S-ESCAPE*⟩	=	`"’"` | `\"` | `"?"` | `"\"` | `"\a"` | `"\b"` | `"\f"` | `"\n"` | `"\r"` | `"\t"` | `"\v"`
⟨*DIGIT*⟩	=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
⟨*UNSIGNED_INTEGER*⟩	=	⟨*DIGIT*⟩ `{` ⟨*DIGIT*⟩ `}`
⟨*UNSIGNED_NUMBER*⟩	=	⟨*UNSIGNED_INTEGER*⟩ [`"."` [⟨*UNSIGNED_INTEGER*⟩]] [(`"e"` | `"E"`) [`"+"` | `"-"`] ⟨*UNSIGNED_INTEGER*⟩]

- *μ*-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.
- Boldface denotes keywords of the *μ*-Modelica language. Keywords are reserved words and may not be used as identifiers.

10.2 Stored Definition

⟨*stored_definition*⟩ ::= `{` ⟨*class_definition*⟩ `";"` `}`

10.3 Class Definition

```
⟨class_definition⟩ ::= ⟨class_prefixes⟩ ⟨class_specifier⟩

⟨class_prefixes⟩   ::= model
                     | package
                     | [ ( pure | impure ) ] function

⟨class_specifier⟩  ::= ⟨IDENT⟩ ⟨string_comment⟩ ⟨composition⟩ end ⟨IDENT⟩

⟨composition⟩      ::= ⟨element_list⟩ { protected ⟨element_list⟩ | ⟨equation_section⟩ | ⟨algorithm_section⟩
                       } [ external [ ⟨language_specification⟩ ] [ ⟨external_function_call⟩ ] [
                       annotation ] ";" ] [ annotation ";" ]

⟨language_specification⟩ ::= STRING

⟨external_function_call⟩ ::= [ ⟨component_reference⟩ "=" ] ⟨IDENT⟩ ( [ ⟨expression_list⟩ ] )

⟨element_list⟩     ::= { ⟨element⟩ ";" }

⟨element⟩          ::= ⟨import_clause⟩
                     | ⟨class_definition⟩
                     | ⟨component_clause⟩

⟨import_clause⟩    ::= import ⟨name⟩
```

10.4 Component Clause

```
⟨component_clause⟩ ::= ⟨type_prefix⟩ ⟨type_specifier⟩ [ ⟨array_subscripts⟩ ] ⟨component_list⟩

⟨type_prefix⟩      ::= [ discrete | parameter | constant ] [ input | output ]

⟨type_specifier⟩   ::= Real
                     | Integer
                     | Boolean

⟨component_list⟩   ::= ⟨component_declaration⟩ { "," ⟨component_declaration⟩ }

⟨component_declaration⟩ ::= ⟨declaration⟩

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

10.5 Modification

```
⟨modification⟩     ::= ⟨class_modification⟩ [ "=" ⟨expression⟩ ]
                     | "=" ⟨expression⟩
                     | ":=" ⟨expression⟩

⟨class_modification⟩ ::= "(" [ ⟨argument_list⟩ ] ")"

⟨argument_list⟩    ::= ⟨argument⟩ { "," ⟨argument⟩ }

⟨argument⟩         ::= ⟨element_modification_or_replaceable⟩

⟨element_modification_or_replaceable⟩ ::= [ each ] ⟨element_modification⟩

⟨element_modification⟩ ::= ⟨name⟩ [ ⟨modification⟩ ]
```

10.6 Equations

```
⟨equation_section⟩ ::= equation { ⟨equation⟩ ";" }

⟨algorithm_section⟩ ::= [ initial ] algorithm { ⟨statement⟩ ";" }
```

```
⟨equation⟩           ::= ⟨simple_expression⟩ "=" ⟨expression⟩
                       | ⟨for_equation⟩
                       | ⟨name function_call_args⟩

⟨statement⟩          ::= ⟨component_reference⟩ ( ":=" ⟨expression⟩ | ⟨function_call_args⟩ )
                       | "(" ⟨output_expression_list⟩ ")" ":=" ⟨component_reference⟩ ⟨function_call_args⟩
                       | return
                       | ⟨if_statement⟩
                       | ⟨for_statement⟩
                       | ⟨when_statement⟩

⟨if_statement⟩       ::= if ⟨expression⟩ then { ⟨statement⟩ ";" } { elseif ⟨expression⟩ then {
                         ⟨statement⟩ ";" } [ else { ⟨statement⟩ ";" } ] end if

⟨for_equation⟩       ::= for ⟨for_index⟩ loop { ⟨equation⟩ ";" } end for

⟨for_statement⟩      ::= for ⟨for_index⟩ loop { ⟨statement⟩ ";" } end for

⟨for_index⟩          ::= ⟨IDENT⟩ in expression

⟨when_statement⟩     ::= when ⟨expression⟩ then { ⟨statement⟩ ";" } { elsewhen ⟨expression⟩ then
                         { ⟨statement⟩ ";" } } end when
```

10.7 Expressions

```
⟨expression⟩         ::= ⟨simple_expression⟩

⟨simple_expression⟩  ::= ⟨logical_expression⟩ [ ":" ⟨logical_expression⟩ [ ":" ⟨logical_expression⟩
                         ] ]

⟨logical_expression⟩ ::= ⟨logical_term⟩ { or ⟨logical_term⟩ }

⟨logical_term⟩       ::= ⟨logical_factor⟩ { and ⟨logical_factor⟩ }

⟨logical_factor⟩     ::= [ not ] ⟨relation⟩

⟨relation⟩           ::= ⟨arithmetic_expression⟩ [ ⟨rel_op⟩ ⟨arithmetic_expression⟩ ]

⟨rel_op⟩             ::= "<"
                       | "<="
                       | ">"
                       | ">="
                       | "=="
                       | "<>"

⟨arithmetic_expression⟩ ::= [ ⟨add_op⟩ ] ⟨term⟩ { ⟨add_op⟩ ⟨term⟩ }

⟨add_op⟩             ::= "+"
                       | "-"

⟨term⟩               ::= ⟨factor⟩ { ⟨mul_op⟩ ⟨factor⟩ }

⟨mul_op⟩             ::= "*"
                       | "/"

⟨factor⟩             ::= ⟨primary⟩ [ "^" ⟨primary⟩ ]

⟨primary⟩            ::= UNSIGNED_NUMBER
                       | false
                       | true
                       | ( name | der ) ⟨function_call_args⟩
                       | ⟨component_reference⟩
                       | "(" ⟨output_expression_list⟩ ")"

⟨name⟩               ::= ⟨IDENT⟩

⟨component_reference⟩ ::= ⟨IDENT⟩ [ ⟨array_subscripts⟩ ]
```

```
⟨function_call_args⟩ ::= "(" [ ⟨function_arguments⟩ ] ")"
⟨function_arguments⟩ ::= ⟨function_argument⟩ [ "," ⟨function_arguments⟩ ]
⟨function_argument⟩ ::= ⟨expression⟩
⟨output_expression_list⟩ ::= [ ⟨expression⟩ ] { "," [ ⟨expression⟩ ] }
⟨expression_list⟩   ::= ⟨expression⟩ { "," ⟨expression⟩ }
⟨array_subscripts⟩ ::= "[" ⟨subscript⟩ { "," ⟨subscript⟩ } "]"
⟨subscript⟩         ::= ⟨expression⟩
⟨annotation⟩        ::= annotation ⟨class_modification⟩
```