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

Language Specification

Version 2.1

January 30, 2004

by the

Modelica Association

Abstract

This document defines the Modelica language, version 2.1, which is developed by the Modelica Association, a non-profit organization with seat in Linköping, Sweden. Modelica is a freely available, object-oriented language for modeling of large, complex, and heterogeneous physical systems. It is suited for multi-domain modeling, for example, mechatronic models in robotics, automotive and aerospace applications involving mechanical, electrical, hydraulic and control subsystems, process oriented applications and generation and distribution of electric power. Models in Modelica are mathematically described by differential, algebraic and discrete equations. No particular variable needs to be solved for manually. A Modelica tool will have enough information to decide that automatically. Modelica is designed such that available, specialized algorithms can be utilized to enable efficient handling of large models having more than one hundred thousand equations. Modelica is suited and used for hardware-in-the-loop simulations and for embedded control systems. More information is available at **http://www.Modelica.org/**

Contents

1 Introduction

1.1 Overview of Modelica

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

1.2 Scope of the specification

The semantics of the Modelica language is specified by means of a set of rules for translating a model described in the Modelica language to the corresponding model described as a flat hybrid DAE. The key issues of the translation (or instantiation in object-oriented terminology) are:

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

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).

The Modelica specification does not define the result of simulating a model or what constitutes a mathematically well-defined model.

1.3 Definitions and glossary

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.*

Term	Definition
Component	An element defined by the production component-clause in the Modelica grammar.
Element	Class definitions, extends-clauses and component-clauses declared in a class.
Instantiation	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** statements.

2 Modelica syntax

2.1 Lexical conventions

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

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

The following lexical units are defined:

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

[The single quotes are part of an identifier. E.g. 'x' and x are different IDENTs].

Note: string constant concatenation "a" "b" becoming "ab" (as in C) is replaced by the "+" operator in Modelica.

Modelica uses the same comment syntax as C++ and Java, and also has structured comments in the form of annotations and string comments. Inside a comment, the sequence `<HTML> .... </HTML>` indicates HTML code which may be used by tools to facilitate model documentation.

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

2.2 Grammar

2.2.1 Stored definition

```
stored_definition:
   [ within [ name ] ";" ]
   { [ final ] class_definition ";" }
```

2.2.2 Class definition

```
class_definition :
   [ encapsulated ]
   [ partial ]
```

```
   ( class | model | record | block | connector | type |
     package | function )
   class_specifier

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

base_prefix :
        type_prefix

enum_list   : enumeration_literal { "," enumeration_literal}

enumeration_literal : IDENT comment

composition  :
   element_list
   { public element_list |
     protected element_list |
     equation_clause |
     algorithm_clause
   }
   [ external [ language_specification ]
              [ external_function_call ] [ annotation ";" ]
              [ annotation  ";" ] ]

language_specification :
   STRING

external_function_call :
   [ component_reference "=" ]
   IDENT "(" [ expression { "," expression } ] ")"

element_list :
   { element ";" | annotation  ";" }

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

```
import_clause :
   import ( IDENT "=" name | name ["." "*"] ) comment
```

2.2.3 Extends

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

constraining_clause :
   extends name [ class_modification ]
```

2.2.4 Component clause

```
component_clause:
   type_prefix type_specifier [ array_subscripts ] component_list

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

type_specifier :
   name

component_list :
   component_declaration { "," component_declaration }

component_declaration :
   declaration comment

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

2.2.5 Modification

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

class_modification :
   "(" [ argument_list ] ")"

argument_list :
   argument { "," argument }

argument :
   element_modification
 | element_redeclaration

element_modification :
   [ each ] [ final ] component_reference [ modification ] string_comment

element_redeclaration :
```

```
   redeclare [ each ] [ final ]
( ( class_definition | component_clause1) |
     replaceable ( class_definition | component_clause1)
        [constraining_clause])

component_clause1 :
   type_prefix type_specifier component_declaration
```

2.2.6 Equations

```
equation_clause :
  [ initial ] equation { equation ";" | annotation  ";" }

algorithm_clause :
  [ initial ] algorithm { algorithm ";" | annotation  ";" }

equation :
  ( simple_expression "=" expression
    | conditional_equation_e
    | for_clause_e
    | connect_clause
    | when_clause_e
    | IDENT function_call )
  comment

algorithm :
  ( component_reference ( ":=" expression | function_call )
    | "(" output_expression_list ")" ":=" component_reference function_call
    | break
    | return
    | conditional_equation_a
    | for_clause_a
    | while_clause
    | when_clause_a )
  comment

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

conditional_equation_a :
   if expression then
     { algorithm ";" }
   { elseif expression then
```

```
      { algorithm ";" }
    }
    [ else
      { algorithm ";" }
    ]
    end if

for_clause_e :
  for for_indices loop
     { equation ";" }
  end for

for_clause_a :
  for for_indices loop
     { algorithm ";" }
  end for

for_indices :
  for_index {"," for_index}

for_index:
  IDENT [ in expression ]

while_clause :
  while expression loop
    { algorithm ";" }
  end while

when_clause_e :
  when expression then
    { equation ";" }
  { elsewhen expression then
    { equation ";" } }
  end when

when_clause_a :
  when expression then
    { algorithm ";" }
  { elsewhen expression then
    { algorithm ";" } }
  end when

connect_clause :
  connect "(" component_reference "," component_reference ")"
```

2.2.7 Expressions

```
expression :
   simple_expression
```

```
 | if expression then expression { elseif expression then expression } else
expression

simple_expression :
   logical_expression [ ":" logical_expression [ ":" logical_expression ] ]

logical_expression :
   logical_term { or logical_term }

logical_term :
   logical_factor { and logical_factor }

logical_factor :
   [ not ] relation

relation :
   arithmetic_expression [ rel_op arithmetic_expression ]

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

arithmetic_expression :
   [ add_op ] term { add_op term }

add_op :
   "+" | "-"

term :
   factor { mul_op factor }

mul_op :
   "*" | "/"

factor :
   primary [ "^" primary ]

primary :
   UNSIGNED_NUMBER
 | STRING
 | false
 | true
 | component_reference [ function_call ]
 | "(" output_expression_list ")"
 | "[" expression_list { ";" expression_list } "]"
 | "{" function_arguments "}"
 | end

name :
   IDENT [ "." name ]
```

```
component_reference :
   IDENT [ array_subscripts ] [ "." component_reference ]

function_call :
   "(" [ function_arguments ] ")"

function_arguments :
   expression [ "," function_arguments | for for_indices ]
 | named_arguments


named_arguments: named_argument [ "," named_arguments ]

named_argument: IDENT "=" expression

output_expression_list:
   [ expression ] { "," [ expression ] }

expression_list :
   expression { "," expression }

array_subscripts :
   "[" subscript { "," subscript } "]"

subscript :
   ":" | expression

comment :
   string_comment [ annotation ]

string_comment :
   [ STRING { "+" STRING } ]

annotation :
   annotation class_modification
```

3 Modelica semantics

3.1 Fundamentals

Instantiation is made in a context which consists of an environment and an ordered set of parents.

3.1.1 Scoping and name lookup

3.1.1.1 Parents

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

Enclosing all class definitions is an unnamed parent that contains all top-level class definitions, and not-yet read classes defined externally as described in section 3.1.4. The order of top-level class definitions in the unnamed parent is undefined.

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

[Example:

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

The unnamed parent of class definition C3 contains C1, C2, and C3 in arbitrary order. When instantiating class definition C3, the set of parents of the declaration of `x` *is the partially instantiated class C3 followed by the unnamed parent with C1, C2, and C3. The set of parents of* `z` *is C4, C3 and the unnamed parent in that order.]*

3.1.1.2 Static name lookup

Names are looked up at class instantiation to find names of base classes, component types, etc. Implicitly defined names of record constructor functions are ignored during type name lookup *[since a record and the implicitly created record constructor function, see section 3.4.8.3, have the same name]*. Names of record classes are ignored during function name lookup.

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

- First look for implicitly declared iteration variables if inside the body of a for-loop, section 3.3.3, or if inside the body of a reduction expression, section 3.4.3.1.

- When an element, equation or algorithm is instantiated, any name is looked up sequentially in each member of the ordered set of parents until a match is found or a parent is encapsulated. In the latter case the lookup stops except for the predefined types, functions and operators defined in this specification. For these cases the lookup continues in the global scope, where they are defined. *[E.g. abs is searched upwards in the hierarchy as usual. If an encapsulated boundary is reached, abs is searched in the global scope instead. The operator abs cannot be redefined in the global scope, because an existing class cannot be redefined at the same level.]*
  Reference to variables successfully looked up in an enclosing parent class is only allowed for variables declared as **constant**. If the use is in a modifier on a short class definition, see section 3.2.7.
  [Example:
  ```
  package A
    constant Real x=2;
    model B
      Real x;
      function foo
        output Real y
      algorithm y:=x; // Illegal since reference to non-constant x in B.]
  ```

- This lookup in each scope is performed as follows

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

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

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

- The first identifier *[A]* is looked up as defined above.
- If the first identifier denotes a component, the rest of the name *[e.g., B or B.C]* is looked up among the declared named component elements of the component.
- If the identifier denotes a class, that class is temporarily instantiated with an empty environment (i.e. no modifiers, see section 3.1.2) and using the parents of the denoted class. The rest of the name [e.g., B or B.C] is looked up among the declared named elements of the temporary instantiated class. If the class does not satisfy the requirements for a package, the lookup is restricted to encapsulated elements only.

[The temporary class instantiation performed for composite names follow the same rules as class instantiation of the base class in an extends clause, local classes and the type in a component clause, except that the environment is empty.]

Lookup of the name of an imported package or class, e.g. A.B.C in the statements **import** A.B.C; **import** D=A.B.C; **import** A.B.C.*, deviates from the normal lexical lookup by starting the lexical lookup of the first part of the name at the top-level.

Qualified import statements may only refer to packages or elements of packages, i.e. in "import A.B.C;" or "import D=A.B.C" A.B must be a package. Unqualified import statements may only import from packages, i.e.

in "import A.B.*;" A.B must be a package. [Note, "import A;" A can be any class as element of the unnamed top-level package]

3.1.1.3 Dynamic name lookup

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

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

[Simple Example:

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

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

More complicated example:

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

class E
   inner Real TI;
   class F
      inner Real TI;
      class G
```

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

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

]

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

[Example:

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

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

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

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

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

class C
```

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

]

3.1.2 Environment and modification

3.1.2.1 Environment

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

3.1.2.2 Merging of modifications

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

[The following larger example demonstrates several aspects:

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

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

```
   class C1
     parameter Real a;
   end C1;
   class C2
     parameter Real b,c;
   end C2;
   class C3
     parameter Real x1;       // No default value
```

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

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

An instantiation of class C4 will give an object with the following variables:

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

]

3.1.2.3 Single modification

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

[Example:

```
  class C1
    Real x[3];
  end C1;
  class C2 = C1(x=ones(3), x[2]=2);  // Error: x[2] designated twice
  class C3
    class C4
      Real x;
    end C4;
    C4 a(x.unit = "V", x.displayUnit="mV",  x=5.0);
  // Ok, different attributes designated (unit, displayUnit and value)
  // identical to:
```

```
    C4 b(x(unit = "V", displayUnit="mV") = 5.0));
  end C3;
```

]

3.1.2.4 Instantiation order

The name of a declared element shall not have the same name as any other element in its partially instantiated parent class. A component shall not have the same name as its type specifier.

Variables and classes can be used before they are declared.

[In fact, declaration order is only significant for:

- *Functions with more than one input variable called with positional arguments, section 3.4.8.*
- *Functions with more than one output variable, section 3.4.8.*
- *Records that are used as arguments to external functions, section 6.2.3*
- *Enumeration literal order within enumeration types, section 3.2.7.1.*

]

In order to guarantee that elements can be used before they are declared and that elements do not depend on the order of their declaration in the parent class, the instantiation proceeds in the following steps:

Flattening

First the names of declared local classes and components are found. Here modifiers are merged to the local elements and redeclarations take effect. Then base-classes are looked up, flattened and inserted into the class. The lookup of the base-classes should be independent *[The lookup of the names of extended classes should give the same result before and after flattening the extends clauses. One should not find any element used during this flattening by lookup through the extends clauses. It should be possible to flatten all extends clauses in a class before inserting the result of flattening. Local classes used for extends should be possible to flatten before inserting the result of flattening the extends clauses.]*

Instantiation

Flatten the class, apply the modifiers and instantiate all local elements.

Check of flattening

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

Modifiers for array elements

The **each** keyword on a modifier requires that it is applied in an array declaration/modification, and the modifier is applied individually to each element of the array. If the modified element is a vector and the modifier does not contain the each-attribute, the modification is split such that the first element in the vector is applied to the first element of the vector of elements, the second to the second element, etc. Matrices and general arrays of elements are treated by viewing those as a vectors of vectors etc.

If the modified element is a vector with subscripts the subscripts must be Integer literals.

If a nested modifier is split, the split is propagated to all elements of the nested modifier, and if they are modified by the each-keyword the split is inhibited for those elements. If the nested modifier that is split in this way

contains re-declarations that are split it is illegal.

[Example:

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

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

3.1.3 Subtyping and type equivalence

3.1.3.1 Subtyping of classes

For any classes S and C, S is a supertype of C and C is a subtype of S if they are equivalent or if:

- every public declaration element of S also exists in C (according to their names)
- those element types in S are supertypes of the corresponding element types in C.

A base class is the class referred to in an extends clause. The class containing the extends clause is called the derived class. *[Base classes of C are typically supertypes of C, but other classes not related by inheritance can also be supertypes of C.]*

3.1.3.2 Subtyping of components

Component B is subtype of A if:

- Both are scalars or arrays with the same number of dimensions
- The type of B is subtype of the base type of A (base type for arrays)
- For every dimension of an array
  - The size of A is indefinite, or
  - The value of expression (size of B) - (size of A) is constant equal to 0 (in the environment of B)

3.1.3.3 Type equivalence

Two types T and U are equivalent if:

- T and U denote the same built-in type (one of RealType, IntegerType, StringType or BooleanType), or
- T and U are classes, T and U contain the same public declaration elements (according to their names), and the elements types in T are *equivalent* to the corresponding element types in U.

3.1.3.4 Type identity

Two elements T and U are identical if:

- T and U are equivalent,
- they are either both declared as **final** or none is declared **final**,

- for a component their type prefixes (see section 3.2.1) are identical, and
- if T and U are classes, T and U contain the same public declaration elements (according to their names), and the elements in T are *identical* to the corresponding element in U.

3.1.3.5 Ordered type identity

Two elements T and U are ordered type identical if and only if:

- T and U are type identical
- If T and U are classes
  - T and U have the same number of elements
  - The i:th declaration element of T and the i:th declaration element of U are ordered type identical

3.1.3.6 Function Type Identity

Two functions T and U have identical type if and only if

- T and U have the same number of input and output elements
- For each input or output element
  - The corresponding elements have the same name
  - The corresponding elements are ordered type identical

3.1.3.7 Enumeration Type Equivalence

Two enumeration types S and E are equivalent if:

- S and E have the same number of enumeration literals
- the i:th enumeration literal of S has the same name as the i:th enumeration literal of E.

3.1.3.8 Subtyping of enumeration types

For any enumeration types S and E, S is a supertype of E and E is a subtype of S if they are equivalent or if S is the type enumeration(:).

[Example:

```
E1 = enumeration(one,two,three);        // E1 not subtype of E2
E2 = enumeration(one,two,three,four);   // E2 is not a subtype of E1
E3 = enumeration(one,two,three,four);   // E3 subtype of E2
E4 = enumeration(:);                    // E1, E2, E3 subtypes of E4
```

•

3.1.4 External representation of classes

Classes may be represented in the hierarchical structure of the operating system *[the file system or a database]*. For classes with version information see also section 7.4.3. The nature of such an external entity falls into one of the following two groups:

- Structured entities *[e.g. a directory in the file system]*
- Non-structured entities *[e.g. a file in the file system]*

3.1.4.1 Structured entities

A structured entity *[e.g. the directory* `A`*]* shall contain a node. In a file hierarchy, the node shall be stored in file `package.mo`. The node shall contain a stored-definition that defines a class *[*`A`*]* with a name matching the name of the structured entity. *[The node typically contains documentation and graphical information for a package, but may also contain additional elements of the class* `A`*.]*

A structured entity may also contain one or more sub-entities (structured or non-structured). The sub-entities are mapped as elements of the class defined by their enclosing structured entity. *[For example, if directory* `A` *contains the three files* `package.mo`*,* `B.mo` *and* `C.mo` *the classes defined are* `A`*,* `A.B`*, and* `A.C`*.]* Two sub-entities shall not define classes with identical names *[for example, a directory shall not contain both the sub-directory* `A` *and the file* `A.mo`*]*.

3.1.4.2 Non-structured entities

A non-structured entity *[e.g. the file* `A.mo`*]* shall contain only a model-definition that defines a class *[*`A`*]* with a name matching the name of the non-structured entity.

3.1.4.3 Within clause

A non-top level entity shall begin with a within-clause which for the class defined in the entity specifies the location in the Modelica class hierarchy. A top-level class may contain a within-clause with no name.

For a sub-entity of an enclosing structured entity, the within-clause shall designate the class of the enclosing entity.

3.1.4.4 Use of MODELICAPATH

The top-level scope implicitly contains a number of classes stored externally. If a top-level name is not found at global scope, a Modelica translator shall look up additional classes in an ordered list of library roots, called MODELICAPATH. *[On a typical system, MODELICAPATH is an environment variable containing a semicolon-separated list of directory names.]*

[The first part of the path `A.B.C` *(i.e.,* `A`*) is located by searching the ordered list of roots in MODELICAPATH. If no root contains* `A` *the lookup fails. If* `A` *has been found in one of the roots, the rest of the path is located in* `A`*; if that fails, the entire lookup fails without searching for* `A` *in any of the remaining roots in MODELICAPATH.]*

3.2 Declarations

3.2.1 Component clause

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

If the type specifier of the component does not denote a built-in type, the name of the type is looked up (3.1.1). The found type is instantiated with a new environment and the partially instantiated parent of the component. It is an error if the type is partial. The new environment is the result of merging

- the modification of parent element-modification with the same name as the component

- the modification of the component declaration

in that order.

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

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

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

Type prefixes (i.e., flow, discrete, parameter, constant, input, output) shall only be applied for type, record and connector components. The type prefixes flow, input and output of a structured component are also applied to the elements of the component. The type prefixes flow, input and output shall only be applied for a structured component, if no element of the component has a corresponding type prefix of the same category. *[For example, input can only be used, if none of the elements has an input or output type prefix]*. The corresponding rules for the type prefixes discrete, parameter and constant are described in section 3.2.2.1.

The rules for components of function types and components in functions are described in section 3.2.13.

3.2.2 Variability prefix

The prefixes **discrete**, **parameter**, **constant** of a component declaration are called *variability prefixes* and define in which situation the variable values of a component are initialized (see section 3.5) 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 3.5).
- A **continuous-time** variable vn may have a non-vanishing time derivative (**der**(vn)≠0 possible) and may change its value at any time during transient analysis (see section 3.5).

If a Real variable is declared with the prefix **discrete** it must be assigned in a when-clause, either by an assignment or an equation.

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

The default variability for Integer, String, Boolean, or enumeration variables is discrete-time, and it is not possible to declare continuous-time Integer, String, Boolean, or enumeration variables. *[A Modelica translator is able to guarantee this property due to restrictions imposed on discrete expressions, see section 3.4.9]*

The variability of expressions and restrictions on variability for definition equations is given in section 3.4.9.

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

identical to zero). Furthermore, memory requirements can be reduced in the simulation environment, if it is known that a component can only change at event instants.

*A **parameter** variable is constant during simulation. This prefix gives the library designer the possibility to express that the physical equations in a library are only valid if some of the used components are constant during simulation. The same also holds for **discrete-time** and **constant** variables. Additionally, the **parameter** prefix allows a convenient graphical user interface in an experiment environment, to support quick changes of the most important constants of a compiled model. In combination with an if-clause, a **parameter** prefix allows to remove parts of a model before the symbolic processing of a model takes place in order to avoid variable causalities in the model (similar to #ifdef in C). Class parameters can be sometimes used as an alternative. Example:*

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

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

```
constant Real PI=4*arctan(1);
```

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

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

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

]

3.2.2.1 Variability of structured entities

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

[Example:

```
record A
```

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

]

3.2.3 Parameter bindings

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

[Example:

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

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

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

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

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

]

3.2.4 Protected elements

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

All elements defined under the heading **protected** are regarded as protected. All other elements *[i.e., defined under the heading public, without headings or in a separate file]* are public *[i.e. not protected]*.

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

3.2.5 Array declarations

The Modelica type system includes scalar number, vector, matrix (number of dimensions, ndim=2), and arrays of more than two dimensions. *[There is no distinguishing between a row and column vector.]*

The following table shows the two possible forms of declarations and defines the terminology. C is a placeholder for any class, including the builtin type classes Real, Integer, Boolean, String, and enumeration types. The type of a dimension upper bound expression, e.g. n, m, p,... in the table below, need to be a subtype of Integer or the name E for an enumeration type E, or Boolean. Colon (:) indicates that the dimension upper bound is unknown and is a subtype of Integer. The lower bound of a dimension is 1 if the dimension index type is a subtype of Integer, or E.e1, if the dimension index type is an enumeration type E=enumeration(e1, …, en), or false if the index type is Boolean. The upper bound is E.en if the dimension index type is the enumeration type E=enumeration(e1, …, en), or true if the index type is Boolean.

An array indexed by Boolean or enumeration type can only be used in the following ways:

- Subscripted using expressions of the appropriate type (i.e. Boolean or the enumerated type)
- Declaration equations of the form x1 = x2 as well as declaration assignments of the form x1 := x2 are allowed for arrays independent of whether the index types of dimensions are subtypes of Integer, Boolean, or enumeration types.

Modelica form 1	Modelica form 2	# dimensions	Designation	Explanation
C x;	C x;	0	Scalar	Scalar
C[n] x;	C x[n];	1	Vector	n – Vector
C[E] x;	C x[E]	1	Vector	Vector index by enumeration type E
C[n, m] x;	C x[n, m];	2	Matrix	n x m Matrix
C[n, m, p, ...] x;	C x[m, n, p ,...];	k	Array	Array with k dimensions (k>=0).

[The number of dimensions and the dimensions sizes are part of the type, and shall be checked for example at redeclarations. Declaration form 1 displays clearly the type of an array, whereas declaration form 2 is the traditional way of array declarations in languages such as Fortran, C, C.

```
Real[:]   v1, v2          // vectors v1 and v2  have unknown sizes. The actual sizes may be different.
```

It is possible to mix the two declaration forms, but it is not recommended

```
Real[3,2] x[4,5];       // x has type  Real[4,5,3,2];
```

A vector y indexed by enumeration values

```
type TwoEnums = enumeration(one,two);
Real[TwoEnums] y;
```

]

Zero-valued dimensions are allowed, so `C x[0];` declares an empty vector and `C x[0,3];` an empty matrix.

[Special cases:

Modelica form 1	Modelica form 2	# dimensions	Designation	Explanation
C[1] x;	C x[1];	1	Vector	1 – Vector, representing a scalar
C[1,1] x;	C x[1, 1];	2	Matrix	1 x 1 – Matrix, representing a scalar
C[n,1] x;	C x[n, 1];	2	Matrix	n x 1 – Matrix, representing a column
C[1,n] x;	C x[1, n];	2	Matrix	1 x n – Matrix, representing a row

]

The type of an array of array is the multidimensional array which is constructed by taking the first dimensions from the component declaration and subsequent dimensions from the maximally expanded component type. A type is maximally expanded, if it is either one of the built-in types (Real, Integer, Boolean, String, enumeration type) or it is not a type class. Before operator overloading is applied, a type class of a variable is maximally expanded.

[Example:

```
type Voltage = Real(unit = "V");
type Current = Real(unit = " A ");
connector Pin
  Voltage      v;          // type class of v = Voltage,  type of v = Real
  flow Current i;          // type class of  i = Current,  type of i = Real
end Pin;
type MultiPin = Pin[5];

MultiPin[4]  p;           // type class of p is MultiPin, type of p is Pin[4,5];

type Point = Real[3];
Point p1[10];
Real  p2[10,3];
```

The components p1 and p2 have identical types.

```
p2[5] = p1[2]+ p2[4];     // equivalent to   p2[5,:] = p1[2,:] + p2[4,:]
Real r[3] = p1[2];        // equivalent to   r[3] = p1[2,:]
```

]

[Automatic assertions at simulation time:

Let A be a declared array and i be the declared maximum dimension size of the d_i-dimension, then an assert statement "assert(i >= 0, …)" is generated provided this assertion cannot be checked at compile time. It is a quality of implementation issue to generate a good error message if the assertion fails.

Let A be a declared array and i be an index accessing an index of the d_i-dimension. Then for every such index-access an assert statement "assert(i >= 1 and i <= size(A,d_i),, …)" is generated, provided this assertion cannot be checked at compile time.

For efficiency reasons, these implicit assert statement may be optionally suppressed.]

3.2.6 Final element modification

An element defined as **final** in an element modification or declaration cannot be modified by a modification or by a redeclaration. All elements of a **final** element are also **final**. *[Setting the value of a parameter in an*

experiment environment is conceptually treated as a modification. This implies that a final modification equation of a parameter cannot be changed in a simulation environment].

[Examples:

```
type Angle = Real(final quantity="Angle", final unit ="rad",
                                          displayUnit="deg");
Angle a1(unit="deg");          // error, since unit declared as final!
Angle a2(displayUnit="rad");   // fine


model TransferFunction
   parameter Real b[:] = {1}   "numerator coefficient vector";
   parameter Real a[:] = {1,1} "denominator coefficient vector";
  ...
end TransferFunction;

model PI "PI controller";
   parameter Real k=1 "gain";
   parameter Real T=1 "time constant";
   TransferFunction tf(final b=k*{T,1}, final a={T,0});
end PI;

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

]

3.2.7 Short class definition

A class definition of the form

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

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

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

[Example: demonstrating the difference in scopes:

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


 model A
   parameter Real R;
```

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

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

A short class definition of the form

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

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

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

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

[Example:

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

the types of `f1` and `f2` are identical.]

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

[Example:

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

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

]

3.2.7.1 Enumeration types

A declaration of the form

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

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

[Example:

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

].

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

[Example:

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

An enumeration type is a simple type and the attributes are defined in section 3.6. The Boolean type name or an enumeration type name can be used to specify the dimension range for a dimension in an array declaration and to specify the range in a for loop range expression. An element of an enumeration type can be accessed in an expression *[e.g. an array index value]*.

[Example:

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

Setting attributes:

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

]

Accessing attribute values in expressions:

```
      Real x[DigitalCurrentChoices];

      // Example using the type name to represent the range
      for e in DigitalCurrentChoices loop
        x[e] := 0.;
      end loop;
```

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

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

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

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

]

3.2.8 Local class definition

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

The uninstantiated local class together with its environment becomes an element of the instantiated parent class.

[The following example demonstrates parameterization of a local class:

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

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

Instantiation of class C2 yields a local class Voltage with nominal-modifier 1000. The variables v1 and v2 instantiate this local class and thus have a nominal value of 1000.]

3.2.9 Extends clause

The name of the base class is looked up in the partially instantiated parent of the extends clause. The found base class is instantiated with a new environment and the partially instantiated parent of the extends clause. The new environment is the result of merging

1. arguments of all parent environments that match names in the instantiated base class
2. the optional class modification of the extends clause

in that order.

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

```
class A
  parameter Real a, b;
end A;
class B
  extends A(b=2);       // Rule #2
end B;
class C
  extends B(a=1);       // Rule #1
end C;
```

]

The elements of the instantiated base class become elements of the instantiated parent class.

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

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

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

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

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

The declaration elements of the instantiated base class shall either

- Not already exist in the partially instantiated parent class *[i.e., have different names]* .
- Be exactly identical to any element of the instantiated parent class with the same name and the same level of protection (**public** or **protected**) and same contents. In this case, one of the elements is ignored (since they are identical it does not matter which one).

Otherwise the model is incorrect.

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

3.2.10 Redeclaration

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

A class declaration of the type 'class extends B(…)' replaces the inherited class B with another declaration that extends the inherited class where the optional class-modification is applied to the inherited class. *[Since this implies that all declarations are inherited with modifications applied there is no need to apply modifiers to the new declaration.]*

For 'class extends B(…)' the inherited class is subject to the same restrictions as a redeclare of the inherited element and the new element is only replaceable if the new definition is replaceable.

[Example:

```
class A
  parameter Real x;
end A;
class B
  parameter Real x=3.14, y;    // B is a subtype of A
end B;
class C
  replaceable A a(x=1);
end C;
class D
  extends C(redeclare B a(y=2));
end D;
```

which effectively yields a class D2 with the contents

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

Example to extend from existing packages:

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

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

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

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

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

  replaceable partial model BaseProperties
    ...
  end BaseProperties;

  replaceable partial function dynamicViscosity
    ...
  end dynamicViscosity;
end PartialMedium;

package Air "Special type of medium"
  extends PartialMedium(nX = 1);

  model extends BaseProperties (T(stateSelect=StateSelect.prefer))
    // extends from Baseproperties with modification
    // note, nX = 1 (!)
    ...
  end BaseProperties

  function extends dynamicViscosity
    // extends from dynamicViscosity
    ...
  end dynamicViscosity;

    or

  redeclare function dynamicViscosity
    // replaces dynamicViscosity by a new implementation
    ...
  end dynamicsViscosity;
end Air;
```

]

3.2.10.1 Constraining type

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

[Example:

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

```
model ElectricalSource
   replaceable Sine source extends MO(final n=5);
   ...
end ElectricalSource;
```

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

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

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

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

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

A redeclaration can redefine the constraining type:

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

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

]

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

In an element redeclaration of a replaceable element the modifiers of the replaced constraining type are merged

to both the new declaration and to the new constraining type, using the normal rules where outer modifiers override inner modifiers.

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

3.2.10.2 Restrictions on redeclarations

The following additional constraints apply to redeclarations:

- only classes and components declared as replaceable can be redeclared with a new type, which must be a subtype of the constraining type of the original declaration, and to allow further redeclarations one must use "**redeclare replaceable**"
- a replaceable class used in an extends clause shall only contain public components *[otherwise, it cannot be guaranteed that a redeclaration keeps the protected variables of the replaceable default class]*
- an element declared as **constant** cannot be redeclared
- an element declared as **parameter** can only be redeclared with **parameter** or **constant**
- an element declared as **discrete** can only be redeclared with **discrete**, **parameter** or **constant**
- a **function** can only be redeclared as **function**
- an element declared as **flow** can only be redeclared with **flow**
- an element declared as not **flow** can only be redeclared without **flow**
- an element declared as **input** can only be redeclared as **input**
- an element declared as **output** can only be redeclared as **output**

Modelica does not allow a protected element to be redeclared as public, or a public element to be redeclared as protected.

Array dimensions may be redeclared.

3.2.10.3 Suggested redeclarations and modifications

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

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

[Example:

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

replaceable Resistor Load(R=2) extends TwoPin
  annotation(choices(
```

```
              choice(redeclare lib2.Resistor Load(a={2}) "…"),
              choice(redeclare Capacitor Load(L=3) "…")));

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

It can also be applied to non-replaceable declarations, e.g. to describe enumerations.

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

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

]

3.2.11 Derivatives of functions

A function declaration can have an annotation derivative specifying the derivative function. This can influence simulation time and accuracy and can be applied to both functions written in Modelica and to external functions. A derivative annotation can state that it is only valid under certain restrictions on the input arguments. These restrictions are defined using the following optional attributes: *order* (only a restriction if order>1, the default for order is 1), *noDerivative*, and *zeroDerivative*. The given derivative-function can only be used to compute the derivative of a function call if these restrictions are satisfied. There may be multiple restrictions on the derivative, in which case they must all be satisfied. The restrictions also imply that some derivatives of some inputs are excluded from the call of the derivative (since they are not necessary). A function may supply multiple derivative functions subject to different restrictions.

[Example:

```
function foo0 annotation(derivative=foo1); end foo0;
function foo1 annotation(derivative(order=2)=foo2); end foo1;
function foo2 end foo2;
```

]

The inputs to the derivative function of order 1 are constructed as follows:

First are all inputs to the original function, and after all them we will in order append one derivative for each input containing reals.

The outputs are constructed by starting with an empty list and then in order appending one derivative for each output containing reals.

If the Modelica function call is a nth derivative (n>=1), i.e. this function call has been derived from an (n-1)th derivative, an annotation(order=n+1)=…, specifies the (n+1)th derivative, and the (n+1)th derivative call is constructed as follows:

The input arguments are appened with the (n+1)th derivative, which are constructed in order from the nth order derivatives.

The output arguments are similar to the output argument for the nth derivative, but each output is one higher in derivative order.

[Example: Given the declarations

```
function foo0
  ...
  input Real x;
  input Boolean linear;
  input ...;
  output Real y;
  ...
  annotation(derivative=foo1);
end foo0;

function foo1
  ...
  input Real x;
  input Boolean linear;
  input ...;
  input Real der_x;
  ...
  output Real der_y;
  ...
  annotation(derivative(order=2)=foo2);
end foo1;

function foo2
  ...
  input Real x;
  input Boolean linear;
  input ...;
  input Real der_x;
  ...;
  input Real der_2_x;
  ...
  output Real der_2_y;
  ...
```

the equation

(...,y(t),...)=foo0(...,x(t),b,...);

implies that:

(...,d y(t)/dt,...)=foo1(...,x(t),b,..., ...,d x(t)/dt,...);

(...,d^2 y(t)/dt^2,...)=foo2(...,x(t),b,...,d x(t)/dt,..., ...,d^2 x(t)/dt^2,...);

]

An input or output to the function may be any simple type (Real,Boolean,Integer, String and enumeration types) or a record, provided the record does not contain both reals and non-reals predefined types. The function must have at least one input containing reals. The output list of the derivative function may not be empty.

zeroDerivative=*input_var1*
The derivative function is only valid if *input_var1* is independent of the variables the function call is differentiated with respect to (i.e. that the derivative of *input_var1* is "zero"). The derivative of *input_var1* is excluded from the argument list of the derivative-function.

[Assume that function f takes a matrix and a scalar. Since the matrix argument is usually a parameter expression it is then useful to define the function as follows (the additional derivative = f_general_der is optional and can be used when the derivative of the matrix is non-zero).

```
function f "Simple table lookup"
  input Real x;
  input Real y[:, 2];
  output Real z;
  annotation(derivative(zeroDerivative=y) = f_der, derivative=f_general_der);
algorithm …
end f;

function f_der "Derivative of simple table lookup"
  input Real x;
  input Real y[:, 2];
  input Real x_der;
  output Real z_der;
algorithm …
end f_der;

function f_general_der "Derivative of table lookup taking into account varying tables"
  input Real x;
  input Real y[:, 2];
  input Real x_der;
  input Real y_der[:, 2];
  output Real z_der;
algorithm …
end f_general_der;
```

]

noDerivative(*input_var2 = f*(*input_var1*, ...))

The derivative function is only valid if the input argument *input_var2* is computed as f(input_var1, ...). The derivative of *input_var2* is excluded from the argument list of the derivative-function.

[Assume that function fg is defined as a composition f(x, g(x)). When differentiating f it is useful to give the derivative under the assumption that the second argument is defined in this way:

```
function fg
  input Real x;
```

```
  output Real z;
algorithm
   z := f(x, g(x));
end fg;

function f
   input Real x;
   input Real y;
   output Real z;
   annotation(derivative(noDerivative(y = g(x))) = f_der);
algorithm …
end f;

function f_der
   input Real x;
   input Real x_der;
   input Real y;
   output Real z_der;
algorithm …
end f_der;
```

This is useful if g represents the major computational effort of fg).]

3.2.12 Restricted classes

The keyword **class** can be replaced by one of the following keywords: **record**, **type**, **connector**, **model**, **block**, **package** or **function**. Certain restrictions will then be imposed on the content of such a definition. The following table summarizes the restrictions. The predefined types are described in section 3.6.

record	No equations are allowed in the definition or in any of its components. May not be used in connections. May not contain protected components.
type	May only be extension to the predefined types, enumerations, records or array of type.
connector	No equations are allowed in the definition or in any of its components.
model	May not be used in connections.
block	Fixed causality, input-output block. Each component of an interface must either have causality equal to input or output. May not be used in connections.
package	May only contain declarations of classes and constants.
function	Same restrictions as for block. Additional restrictions: no equations, no initial algorithm, at most one algorithm clause. Calling a function requires either an algorithm clause or an external function interface. A function can not contain calls to the Modelica built-in operators **der, initial, terminal, sample, pre, edge, change, reinit, delay** and **cardinality** and to the operators of the built-in package **Connections**.

3.2.13 Components of function type

A function can be called, 3.4.8, or a component of function type can be instantiated. It is also possible to modify and extend a function class e.g. to add default values for input variables.

When a function is called components of a function do not have start-attributes, but a binding assignment (":=" expression) is an expression such that the component is initialized to this expression at the start of every function invocation (before executing the algorithm section or calling the external function). Binding assignments can only be used for components of a function. If no binding 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. Binding assignment for input arguments are interpreted as default arguments, as described in section 3.4.8. The size of each non-input array component of a function must be given by the inputs. Components of a function will inside the function behave as though they had discrete-time variability.

When instantiating a component of function type it behaves as though the following rules were followed:

- All binding assignments in the function to its components are ignored.
- The algorithm/external section of the function component is replaced by
  ```
  equation
    (out1,out2,…)=function call(inp1=inp1,inp2=inp2,…);
  ```
  where this function call behaves as described above.
- Protected components of the component of function type are ignored, since they are not given a value by the above-mentioned function-call.

[Example:

```
connector InPort = input Real;
connector OutPort = output Real;
function sin
  input  InPort u;
  output OutPort y;
  protected Real x;
  external "C";
  annotation(...);
end sin;
```

It can then be used as:

```
  Real y=sin(time); // Direct call
  sin sin1; // Component of function type
  Clock clock;
equation
  connect(clock.y, sin1.u); // Connect to the object.
  // Can use: sin1.y,
  // Cannot use: sin1.x since x is protected and thus ignored
```

This can be called as a normal function and since InPort/OutPort are connectors we can also connect to components of the function type.

]

3.3 Equations and Algorithms

3.3.1 Equation and Algorithm clauses

The instantiated equation or algorithm is identical to the non-instantiated equation or algorithm.

Names in an equation or algorithm shall be found by looking up in the partially instantiated parent of the equation or algorithm.

Equation equality **=** shall not be used in an algorithm clause. The assignment operator **:=** shall not be used in an equation clause.

3.3.2 If clause

The expression of an **if** and **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. In an equation section, the equations in the body are seen as equations that must be satisfied. The bodies that are not selected have no effect on that model evaluation.

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

3.3.3 For clause

The following construct

```
for IDENT in expression loop
```

is one example of a prefix of a for clause.

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

[Example:

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

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

```
constant Integer j=4;
Real x[j];
```

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

]

3.3.3.1 Deduction of ranges

An iterator `IDENT in range-expr` without the '`in range-expr`' requires that the IDENT appear as *the subscript* of one or several subscripted expressions. The dimension size of the array expression in the indexed position is used to deduce the `range-expr` as 1:size(array-expression,indexpos) if the indices are a subtype of Integer, or as E.`e1:E.en` if the indices are of an enumeration type E=enumation(e1, …, en), or as `false:true` if the indices are of type `Boolean`. If it is used to subscript several expressions, their ranges must be identical. The IDENT may also, inside a reduction-expression, array constructor expression, or for-clause, occur freely outside of subscript positions, but only as a reference to the variable IDENT, and not for deducing ranges.

[Example:

```
  Real x[4];
  Real xsquared[:]={x[i]*x[i] for i};
  // Same as: {x[i]*x[i] for i in 1:size(x,1)}
  Real xsquared2[size(x,1)];
equation
  for i loop // Same as: for i in 1:size(x,1) loop …
    xsquared2[i]=x[i]^2;
  end for;

type FourEnums=enumeration(one,two,three,four);
  Real xe[FourEnums]= x;
  Real xsquared3[FourEnums]={xe[i]*xe[i] for i};
  Real xsquared4[FourEnums]={xe[i]*xe[i] for i in FourEnums};
  Real xsquared5[FourEnums]={x[i]*x[i] for i};
```

]

3.3.3.2 Several iterators

The notation with several iterators is a shorthand notation for nested for-clauses (or reduction-expressions). For for-clauses it can be expanded into the usual form by replacing each "," by '`loop for`' and adding extra '`end for`'. For reduction-expressions it can be expanded into the usual form by replacing each ',' by '`) for`' and prepending the reduction-expression with '`function-name(`'.

[Example:

```
  Real x[4,3];
equation
  for j, i in 1:2 loop
    // The loop-variable j takes the values 1,2,3,4 (due to use)
    // The loop-variable i takes the values 1,2 (given range)
    x[j,i]=j+i;
  end for;
```

]

3.3.4 When clause

The expression of a when clause shall be a discrete-time Boolean scalar or vector expression. The equations and algorithm statements within a when clause are activated when the scalar or any one of the elements of the vector expression becomes true. When-clauses in equation sections are allowed, provided the equations within the when-clause have one of the following forms:

- v = expr;
- (out1, out2, out3, ...) = function_call(in1, in2, ...);
- operators **assert**(), **terminate**(), **reinit**()
- **For** and **if**-clause if the equations within the **for** and **if**-clauses satisfy these requirements.
- In an equation section, the different branches of when/elsewhen must have the same set of component references on the left-hand side.
- In an equation section, the branches of an if-then-else clause inside when-clauses must have the same set of component references on the left-hand side, unless the if-then-else have exclusively parameter expressions as switching conditions.

A when clause shall not be used within a function class.

[Example:

Algorithms are activated when x becomes > 2:

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

Algorithms are activated when either x becomes > 2 or sample(0,2) becomes true or x becomes less than 5:

```
when {x > 2, sample(0,2), x < 5} then
  y1 := sin(x);
  y3 := 2*x + y1+y2;
end when;
```

For when in equation sections the order between the equations does not matter, e.g.

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

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

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

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

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

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

For when in algorithm sections the order is significant and it is advisable to have only one assignment within the when-clause and instead use several algorithms having when-clauses 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-clauses can lead to less efficient code and different models with different behaviour depending on the order of the assignment to y1 and y3 in the algorithm.]

A when clause

```
algorithm
  when {x>1, ..., y>p} then
    ...
  elsewhen x > y.start then
     ...
  end when;
```

is equivalent to the following special if-clause, where Boolean b1[N]; and Boolean b2 are necessary because the **edge**() operator can only be applied to variables

```
  Boolean b1[N](start={x.start>1, ..., y.start>p});
  Boolean b2(start=x.start>y.start);
algorithm
  b1:={x>1, ..., y>p};
  b2:=x>y.start;

  if edge(b1[1]) or edge(b1[2]) or ... edge(b1[N]) then
    ...
  elseif edge(b2) then
    ...
  end if;
```

with "**edge**(A) = A **and not pre**(A)" and the additional guarantee, that the algorithms within this special if clause are only evaluated at event instants.

A when-clause

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

is equivalent to the following special if-expressions

```
  Boolean b(start=x.start>2);
equation
  b  = x>2;
  v1 = if edge(b) then expr1 else pre(v1);
  v2 = if edge(b) then expr2 else pre(v2);
```

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

When clauses cannot be nested.

[Example:

The following when clause is invalid:

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

]

3.3.5 While clause

The expression of a while clause shall be a scalar boolean expression. The while-clause corresponds to while-statements in programming languages, and is formally defined as follows

1. The expression of the while clause is evaluated.
2. If the expression of the while-clause is false, the execution continues after the while-clause.
3. If the expression of the while-clause is true, the entire body of the while clause is executed (except if a break statement, see section 3.3.6, or return statement, see section 3.3.7, is executed), and then execution proceeds at step 1.

3.3.6 Break statement

The break statement breaks the execution of the innermost while or for-loop enclosing the break statement and

continues execution after the while or for-loop. It can only be used in a while or for-loop in an algorithm section.

[Example (note this could alternatively use return):

```
function findValue "Returns position of val or 0 if not found"
  input Integer x[:];
  input Integer val;
  output Integer index;
algorithm
  index := size(x,1);
  while index >= 1 loop
    if x[index]== val then
      break;
    else
      index := index - 1;
    end if;
  end while;
end findValue;
```

]

3.3.7 Return statement

The return statement terminates the current function call, see section 3.4.8. It can only be used in an algorithm section of a function.

[Example (note this could alternatively use break):

```
function findValue "Returns position of val or 0 if not found"
  input Integer x[:];
  input Integer val;
  output Integer index;
algorithm
  for i in 1:size(x,1) loop
    if x[i] == val then
      index := i;
      return;
  end for;
  index := 0;
  return;
end findValue;
```

]

3.3.8 Connections

Connections between objects are introduced by the **connect** statement in the equation part of a class. The **connect** construct takes two references to connectors, each of which is either of the following forms:

- c1.c2. … .cn, where c1 is a connector of the class, n>=1 and ci+1 is a connector element of ci for i=1:(n-1).
- m.c, where m is a non-connector element in the class and c is a connector element of m.

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

[Example of array use:

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

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

]

The two main tasks are to:

- Build connection sets from **connect** statements.
- Generate equations for the complete model.

Definitions:

Connection sets

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

Inside and outside connectors

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

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

3.3.8.1 Generation of connection equations

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

For every use of the connect statement

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

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

```
a1 = a2 = ... = an;
z1 + z2 + (-z3) + ... + zn = 0;
```

In order to generate equations for through variables *[using the `flow` prefix]*, the sign used for the connector variable z_i above is +1 for inside connectors and -1 for outside connectors *[`z3` in the example above]*.

For each flow (zero-sum) variable in a connector that is not connected as an inside connector in any element instance the following equation is implicitly generated:

z = **0**;

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

3.3.8.2 Restrictions

A component of a connector declared with the **input** type prefix shall not occur as inside connector in more than one **connect** statement. A component of a connector declared with the **output** type prefix shall not occur as outside connector in more than one **connect** statement. If two components declared with the **input** type prefix are connected in a **connect** statement one must be an inside connector and the other an outside connector. If two components declared with the **output** type prefix are connected in a **connect** statement one must be an inside connector and the other an outside connector.

Subscripts in a connector reference shall be constant expressions.

If the array sizes do not match, the original variables are filled with one-sized dimensions from the left until the number of dimensions match before the connection set equations are generated.

Constants or parameters in connected components yield the appropriate assert statements; connections are not generated.

3.3.8.3 Overdetermined connection equations and virtual connection graphs

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

This section handles a certain class of overdetermined systems due to connectors that have a redundant set of

variables. There are other causes of overdetermined systems, e.g., explicit zero-sum equations for flow variables, that are not handled by the method described below.]

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

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

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

record Record
   < declaration of record fields>

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

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

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

[if the elements of a record Record are not independent from each other, the equation "R1 = R2" contains redundant equations]. A *type* class with an *equalityConstraint* function declaration is called *overdetermined type*. A record class with an *equalityConstraint* function definition is called *overdetermined record*. A connector that contains instances of overdetermined type and/or record classes is called *overdetermined connector*. An overdetermined type or record may neither have flow components nor may be used as a type of flow components.

Every instance of an *overdetermined type* or *record* in an *overdetermined connector* is a **node** in a *virtual connection graph* that is used to determine when the standard equation "R1 = R2" or when the equation "**0** = equalityConstraint(R1,R2)"has to be used for the generation of connect(...) equations. The **branches** of the virtual connection graph are implicitly defined by "connect(..)" and explicitly by "Connections.branch(..)" statements, see table below. "Connections" is a built-in package in global scope containing built-in operators. Additionally, corresponding nodes of the virtual connection graph have to be defined as **roots** or as **potential**

roots with functions “Connections.root(..)” and “Connections.potentialRoot(..)”, respectively. In the following table, A and B are connector instances that may be **hierarchically** structured, e.g., A may be an abbreviation for “EnginePort.Frame”.

connect(A,B);	Defines **breakable branches** from the overdetermined type or record instances in connector instance A to the corresponding overdetermined type or record instances in connector instance B for a virtual connection graph. The types of the corresponding overdetermined type or record instances shall be the same.
Connections.branch(A.R,B.R);	Defines a **non-breakable branch** from the overdetermined type or record instance R in connector instance A to the corresponding overdetermined type or record instance R in connector instance B for a virtual connection graph. This function can be used at all places where a connect(..) statement is allowed *[e.g., it is not allowed to use this function in a when clause. This definition shall be used if in a model with connectors A and B the overdetermined records A.R and B.R are algebraically coupled in the model, e.g., due to B.R = f(A.R, <other unknowns>)]*.
Connections.root(A.R);	The overdetermined type or record instance R in connector instance A is a (definite) **root node** in a virtual connection graph. *[This definition shall be used if in a model with connector A the overdetermined record A.R is (consistently) assigned, e.g., from a parameter expressions]*
Connections.potentialRoot(A.R); **Connections.potentialRoot** (A.R, priority = p);	The overdetermined type or record instance R in connector instance A is a **potential root node** in a virtual connection graph with priority “p” ($p \geq 0$). If no second argument is provided, the priority is zero. “p” shall be a parameter expression of type Integer. In a virtual connection subgraph without a Connections.root definition, one of the potential roots with the lowest priority number is selected as root *[This definition may be used if in a model with connector A the overdetermined record A.R appears differentiated – **der**(A.R) – together with the **constraint equations** of A.R, i.e., a non-redundant subset of A.R maybe used as states]*
b = **Connections.isRoot**(A.R);	Returns true, if the overdetermined type or record instance R in connector instance A is selected as a root in the virtual connection graph.

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

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

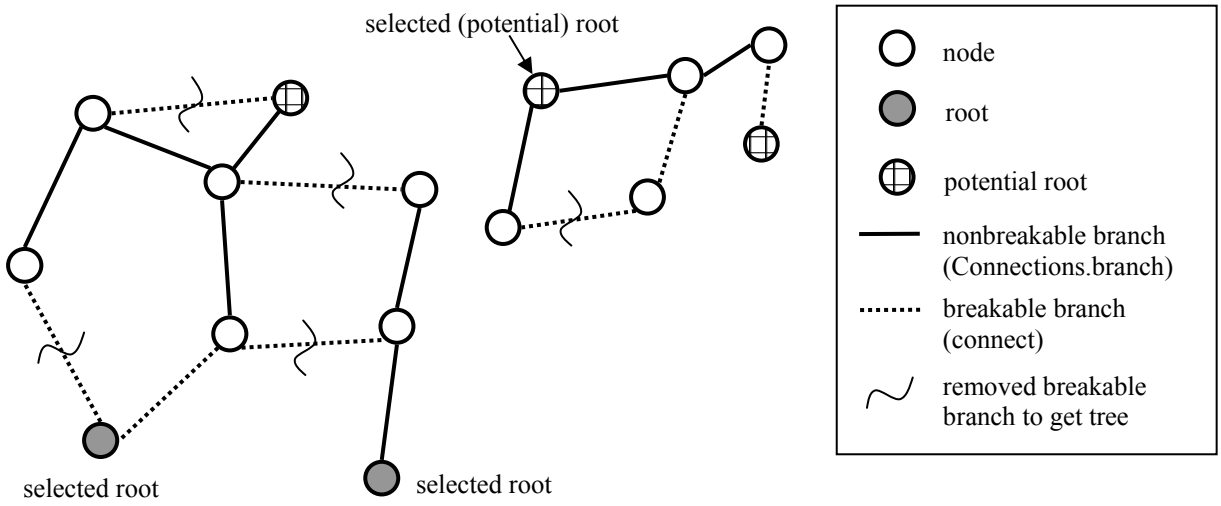
1. Every root node defined via the “Connections.root(..)” statement is a definite root of one spanning tree.
2. The virtual connection graph may consist of sets of subgraphs that are not connected together. Every subgraph in this set shall have at least one root node or one potential root node. If a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

3. If there are n selected roots in a subgraph, then breakable branches have to be removed such that the result shall be a set of n spanning trees with the selected root nodes as roots.

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

1. For every breakable branch *[, i.e., connect(A,B) statement,]* in one of the spanning trees, the connect equations are generated according to section 3.3.8.1.
2. For every breakable branch **not** in any of the spanning trees, the connect equations are generated according to section 3.3.8.1, except for overdetermined type or record instances R. Here the equations "**0** = R.equalityConstraint(A.R,B.R)" are generated instead of "A.R = B.R".

[Example for virtual connection graph:



]

Here are more complete examples for using the operators:

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

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

  function equalityConstraint
    input  AC_Angle theta1;
    input  AC_Angle theta2;
    output Real residue[0] "No constraints"
  algorithm
    /* make sure that theta1 and theta2 from
       joining branches are identical */
```

```
    assert(abs(theta1 - theta2) < 1.e-10);
  end equalityConstraint;
end AC_Angle;

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

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

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

At the place where the source frequency, i.e., essentially variable theta , is defined, a Connections.root(..) must be present:

```
  AC_plug p;
equation
  Connections.root(p.theta);
  der(p.theta) = 2*Modelica.Constants.pi*50   // 50 Hz;
```

The graph analysis performed with the virtual connection graph identifies the connectors, where the AC_Angle needs not to be passed between components, in order to avoid redundant equations.

An overdetermined connector for 3-dimensional mechanical systems may be defined as:

```
type TransformationMatrix = Real[3,3];

type Orientation "Orientation from frame 1 to frame 2"
  extends Real[3,3];

  function equalityConstraint
    input  Orientation R1 "Rotation from inertial frame to frame 1";
    input  Orientation R2 "Rotation from inertial frame to frame 2";
    output Real residue[3];
  protected
```

```
    Orientation R_rel "Relative Rotation from frame 1 to frame 2";
  algorithm
    R_rel = R2*transpose(R1);
    /* If frame_1 and frame_2 are identical, R_rel must be
       the unit matrix. If they are close together, R_rel can be
       linearized yielding:
         R_rel = [    1,  phi3, -phi2;
                  -phi3,     1,  phi1;
                   phi2, -phi1,     1 ];
       where phi1, phi2, phi3 are the small rotation angles around
       axis x, y, z of frame 1 to rotate frame 1 into frame 2
    */
    residue := {R_rel[2, 3], R_rel[3, 1], R_rel[1, 2]};
  end equalityConstraint;
end Orientation;

connector Frame "3-dimensional mechanical connector"
   import SI = Modelica.SIunits;
   SI.Position    r[3] "Vector from inertial frame to Frame";
   Orientation    R    "Orientation from inertial frame to Frame";
   flow SI.Force  f[3] "Cut-force resolved in Frame";
   flow SI.Torque t[3] "Cut-torque resolved in Frame";
end Frame;
```

A fixed translation from a frame A to a frame B may be defined as:

```
model FixedTranslation
  parameter Modelica.SIunits.Position r[3];
  Frame frame_a, frame_b;
equation
  Connections.branch(frame_a.R, frame_b.R);
  frame_b.r = frame_a.r + transpose(frame_a.R)*r;
  frame_b.R = frame_a.R;
  zeros(3)  = frame_a.f + frame_b.f;
  zeros(3)  = frame_a.t + frame_b.t + cross(r, frame_b.f);
end FixedTranslation;
```

Since the transformation matrix frame_a.R is algebraically coupled with frame_b.R, a branch in the virtual connection graph has to be defined. At the inertial system, the orientation is consistently initialized and therefore the orientation in the inertial system connector has to be defined as root:

```
model InertialSystem
  Frame frame_b;
equation
  Connections.root(frame_b.R);
  frame_b.r = zeros(3);
  frame_b.R = identity(3);
end InertialSystem;
```

3.3.9 Initialization

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

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

- As equations in an "**initial equation**" section or as assignments in an "**initial algorithm**" section. The equations and assignments in these initial sections are purely algebraic, stating constraints between the variables at the initial time instant. It is not allowed to use **when**-clauses in these sections.
- Implicitly by using the attributes start=value and fixed=**true** in the declaration of variables:
  For all non-discrete Real variables v, the equation "v = startExpression" is added to the initialization equations, if "start = startExpression" and "fixed = **true**".
  For all discrete variables vd, the equation "**pre**(vd) = startExpression" is added to the initialization equations, if "start = startExpression" and "fixed = **true**.
  For constants and parameters, the attribute fixed is by default **true**. For other variables fixed is by default **false**.

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

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

Examples:

Continuous time controller initialized in steady-state:

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

This has the following solution at initialization:

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

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

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

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

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

Discrete time controller initialized in steady-state:

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

This leads to the following equations during initialization:

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

With the solution:

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

3.4 Expressions

Modelica equations, assignments and declaration equations contain expressions.

Expressions can contain basic operations, +, -, *, /, ^, etc. with normal precedence as defined in the grammar in section 2.2.7. The semantics of the operations is defined for both scalar and array arguments in section 3.4.6.

It is also possible to define functions and call them in a normal fashion. The function call syntax for both normal and named arguments is described in section 3.4.8 and for vectorized calls in section 3.4.6.10. The built-in array functions are given in section 3.4.3 and other built-in operators in section 3.4.2.

3.4.1 Evaluation

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.

[Example. If one wants to guard an expression against 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
```

]

3.4.1.1 Pure functions

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

- Modelica functions are *mathematical functions*, i.e. calls with the same input argument values always give the same results.
- A Modelica function is side-effect free with respect to the internal Modelica simulation state. Specifically, the *ordering* of function calls and the number of calls to a function shall not influence the simulation state.

[Comment 1: This property enables writing declarative specifications using Modelica. It also makes it possible for Modelica compilers to freely perform algebraic manipulation of expressions containing function calls while still preserving their semantics.]

[Comment 2: The Modelica translator is responsible for maintaining this property for pure non-external functions. Regarding external functions, the external function implementer is responsible. Note that external functions can have side-effects as long as they do not influence the internal Modelica simulation state, e.g. caching variables for performance or printing trace output to a log file.]

- Exception: An impure function is either an impure external function or a Modelica function calling an impure function. An impure function (that may return different values at different calls despite having the same input argument values), may be called from within an impure function, from within a when-clause and during initialization.

[Comment:The semantics are undefined if the function call is part of an algebraic loop.

This exceptional case allows, e.g. calling impure external functions at events in when-clauses during hardware-in-the-loop simulation, to obtain input from the external hardware component, and/or provide output to be communicated to such a component.

The above implies that if all external functions that are used are pure then all Modelica functions are pure.]

3.4.2 Modelica built-in operators

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. The following operators are supported (see also the list of array function in section 3.4.3):

der(x)	The time derivative of x. Variable x need to be a subtype of Real, and may not be a discrete-time variable. If x is an array, the operator is applied to all elements of the array. For Real parameters and constants the result is a zero scalar or array of the same size as the variable.
initial()	Returns **true** during the initialization phase and **false** otherwise.
terminal()	Returns true at the end of a successful analysis.
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 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 3.4.2.2.
noEvent(expr)	Real elementary relations within expr are taken literally, i.e., no state or time event is triggered. See also sections 3.4.2.2, and 3.5.
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: (a) variable y is a subtype of a simple type, (b) y is a discrete-time expression (c) the operator is **not** applied in a **function** class. The first value of **pre**(y) is determined in the initialization phase. See also section 3.4.2.1.
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)	Reinitializes state variable x with expr at an event instant. Argument x need to be (a) a subtype of Real and (b) the **der-**operator need to be applied to it. expr need to be an Integer or Real expression. The reinit operator can only be applied once

	for the same variable x. It can only be applied in the body of a when-clause. See also section 3.4.2.3
assert(condition, message)	The condition shall be true for successful model evaluations. For full description see section 3.4.2.4.
terminate(message)	Successfully terminates the current analysis. For full description see section 3.4.2.5.
abs(v)	Is expanded into "(**if** v >= 0 **then** v **else** –v)". Argument v needs to be an Integer or Real expression. *[Note, outside of a when clause state events are triggered]*.
sign(v)	Is expanded into "(**if** v > 0 **then** 1 **else if** v < 0 **then** –1 **else** 0)". Argument v needs to be an Integer or Real expression. *[Note, outside of a when clause state events are triggered]*
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.
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.]*.
Integer(e)	Returns the ordinal number of the enumeration value E.enumvalue, where Integer(E.e1) =1, Integer(E.en) =size(E), for an enumeration type E=enumeration(e1, …, en).

String(b, <options>) **String**(i, <options>) **String**(r, significantDigits=d, <options>) **String**(r, format, <options>) **String**(e, <options>)	Convert a scalar non-String expression to a String representation. The first argument maybe a Boolean b, an Integer i, a Real r or an Enumeration e. 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. If no *format* is given the C format ".Ng" is used, where N = *significant Digits*. The default is N=6, i.e., ".6g". *Integer significantDigits=6*: defines the number of significant digits in the result string. *[Examples: "12.3456", "0.0123456", "12345600", "1.23456E-10"].* *String format*: According to ANSI-C without "%" character and "*" is not supported *[Examples: ".6g", "14.5e", "+6f"].*
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 3.4.2.7.
cardinality(c)	Returns the number of (inside and outside) occurrences of connector instance c in a connect statement as an Integer number. See also section 3.4.2.8.
isPresent(ident)	Boolean isPresent(*ident*) returns true if the formal input or output argument *ident* is present as an actual argument of the function call. If the argument is not present, isPresent(*ident*) may return false *[but may also return true e.g. for implementations that always compute all results]*. isPresent() should be used for optimisation only and should not influence the results of outputs that are present in the output list. It can only be used in functions.
semiLinear(x, positiveSlope, negativeSlope)	Returns "if x >= 0 then positiveSlope*x else negativeSlope*x". The result is of type Real. See section 3.4.2.9 *[especially in the case when x = 0]*. For non-scalar arguments the function is vectorized according to section 3.4.6.11.

3.4.2.1 pre

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

*[If v and **pre**(v) are only used in when clauses, the translator might mask event iteration for variable v since v cannot change during event iteration. It is a "quality of implementation" to find the minimal loops for event iteration, i.e., not all parts of the model need to be reevaluated.*
The language allows mixed algebraic systems of equations where the unknown variables are of type Real, Integer, Boolean, or an enumeration. These systems of equations can be solved by a global fix point iteration

scheme, similarly to the event iteration, by fixing the Boolean, Integer, and/or enumeration unknowns during one iteration. Again, it is a quality of implementation to solve these systems more efficiently, e.g., by applying the fix point iteration scheme to a subset of the model equations.]

3.4.2.2 noEvent and smooth

The noEvent operator implies that real elementary expressions are taken literally instead of generating crossing functions, section 3.5. The smooth operator should be used instead of noEvent, in order to avoid events for efficiency reasons. A tool is free to not generate events for expressions inside smooth. However, smooth does not guarantee that no events will be generated, and thus it can be necessary to use noEvent inside smooth. *[Note that smooth does not guarantee a smooth output if any of the occurring variables change discontinuously.]*
[Example:

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

]

3.4.2.3 reinit

The **reinit** operator does not break the single assignment rule, because **reinit**(x,expr) makes the previously known state variable x unknown and introduces the equation "x = expr".

*[If a higher index system is present, i.e. constraints between state variables, some state variables need to be redefined to non-state variables. If possible, non-state variables should be chosen in such a way that states with an applied **reinit** operator are not utilized. If this is not possible, an error occurs, because the reinit operator is applied on a non-state variable.*

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

```
Bouncing ball:
    der(h) = v;
    der(v) = -g;

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

]

3.4.2.4 assert

A statement

```
  assert(condition, message);
```

is an assert-statement, where condition is a boolean expression and message is a string expression.

If the condition of an assert statement is true, message is not evaluated and the procedure call is ignored. If the condition evaluates to false the current evaluation is aborted. The simulation may continue with another evaluation.

Failed assertions takes precedence over successful termination, such that if the model first triggers the end of successful analysis by reaching the stop-time or explicitly with terminate(), but the evaluation with

terminal()=true triggers an assert, the analysis failed. *[The intent is to perform a test of model validity and to report the failed assertion to the user if the expression evaluates to false. The means of reporting a failed assertion are dependent on the simulation environment. The intention is that the current evaluation of the model should stop when an assert with a false condition is encountered, but the tool should continue the current analysis (e.g. by using a shorter stepsize).]*

3.4.2.5 terminate

The terminate function successfully terminates the analysis which was carried out, see also section 3.4.2.4. The function has a string argument indicating the reason for the success. *[The intention is to give more complex stopping criteria than a fixed point in time. Example:*

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

]

3.4.2.6 Event triggering operators

*[The **div**, **rem**, **mod**, **ceil**, **floor**, **integer**, **abs** and **sign** operator trigger state events if used outside of a **when** clause. If this is not desired, the **noEvent** function can be applied to them. E.g. **noEvent(abs**(v)) is |v|*

3.4.2.7 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 need 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 realtime 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 principal, a delay operator could break algebraic loops. For simplicity, this is not supported because the minimum delay time has to be give as additional argument to be fixed at compile time. Furthermore, the maximum step size of the integrator is limited by this minimum delay time in order to avoid extrapolation in the delay buffer.*

3.4.2.8 cardinality

*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
   // Handle cases if pins are not connected
      if cardinality(p) == 0 and cardinality(n) == 0 then
         p.v = 0; n.v = 0;
      elseif cardinality(p) == 0 then
         p.i = 0;
      elseif cardinality(n) == 0 then
         n.i = 0;
      end if;

   // Equations of resistor
      ...
end Resistor;
```

]

3.4.2.9 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**, s1);
y = semiLinear(x, s1, s2);
y = semiLinear(x, s2, s3);
...
y = semiLinear(x, s_N, **sb**);
....

may be replaced by

s1 = **if** m_dot >= 0 **then** sa **else** sb
s2 = s1;
s3 = s2;
...
$s_N = s_{N-1}$;
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_dot, port_h, h);

is identical to

-semiLinear(-m_dot, h, port_h);

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

H_dot =semiLinear(m_dot, port.h, h);

i.e., the enthalpy flow rate H _dot is computed from the mass flow rate m_dot and the upstream specific enthalpy depending on the flow direction.

]

3.4.3 Vectors, Matrices, and Arrays Built-in Functions for Array Expressions

The following function *cannot* be used in Modelica, but is utilized below to define other operators

promote(A,n)	Fills dimensions of size 1 from the right to array A upto dimension n, where "n >= ndims(A)" is required. Let C = promote(A,n), with nA=ndims(A), then ndims(C) = n, size(C,j) = size(A,j) for 1 <= j <= nA, size(C,j) = 1 for nA+1 <= j <= n, C[i_1, ..., i_nA, 1, ..., 1] = A[i_1, ..., i_nA]

[Function promote could not be used in Modelica, because the number of dimensions of the return array cannot be determined at compile time if n is a variable. Below, promote is only used for constant n].

The following built-in functions for array *expressions* are provided:

Modelica	*Explanation*
ndims(A)	Returns the number of dimensions k of array expression A, with k >= 0.
size(A,i)	Returns the size of dimension i of array expression A where i shall be > 0 and <= ndims(A).
size(A)	Returns a vector of length ndims(A) containing the dimension sizes of A.
scalar(A)	Returns the single element of array A. size(A,i) = 1 is required for 1 <= i <= ndims(A).
vector(A)	Returns a 1-vector, if A is a scalar and otherwise returns a vector containing all the elements of the array, provided there is at most one dimension size > 1.
matrix(A)	Returns promote(A,2), if A is a scalar or vector and otherwise returns the elements of the first two dimensions as a matrix. size(A,i) = 1 is required for 2 < i <= ndims(A).
transpose(A)	Permutes the first two dimensions of array A. It is an error, if array A does not have at least 2 dimensions.

outerProduct(v1,v2)	Returns the outer product of vectors v1 and v2 (= matrix(v)*transpose(matrix(v))).
identity(n)	Returns the n x n Integer identity matrix, with ones on the diagonal and zeros at the other places.
diagonal(v)	Returns a square matrix with the elements of vector v on the diagonal and all other elements zero.
zeros(n_1,n_2,n_3,...)	Returns the n_1 x n_2 x n_3 x ... Integer array with all elements equal to zero (n_i >= 0).
ones(n_1,n_2,n_3,...)	Return the n_1 x n_2 x n_3 x ... Integer array with all elements equal to one (n_i >=0).
fill(s,n_1,n_2,n_3, ...)	Returns the n_1 x n_2 x n_3 x ... array with all elements equal to scalar or array expression s (n_i >= 0). The returned array has the same type as s. Recursive definition: **fill**(s,n_1,n_2,n_3, ...)= **fill**(**fill**(s,n_2,n_3, ...), n_1); **fill**(s,n)={s,s,…, s}
linspace(x1,x2,n)	Returns a Real vector with n equally spaced elements, such that v=linspace(x1,x2,n), v[i] = x1 + (x2-x1)*(i-1)/(n-1) for 1 <= i <= n. It is required that n >= 2. The arguments x1 and x2 shall be numeric scalar expressions.
min(A)	Returns the smallest element of array expression A.
min(x,y)	Returns the smallest element of the scalars x and y.
min(e(i, ..., j) **for** i **in** u, ..., j **in** v)	Described in section 3.4.3.1. Returns the smallest value of the scalar expression e(i, ..., j) evaluated for all combinations of i in u, ..., j in v:
max(A)	Returns the largest element of array expression A.
max(x,y)	Returns the largest element of the scalars x and y.
max(e(i, ..., j) **for** i **in** u, ..., j **in** v)	Described in section 3.4.3.1. Returns the largest value of the scalar expression e(i, ..., j) evaluated for all combinations of i in u, ..., j in v:
sum(A)	Returns the scalar sum of all the elements of array expression: A[1,...,1]+A[2,...,1]+....+A[end,...,1]+A[end,...,end]
sum(e(i, ..., j) **for** i **in** u, ..., j **in** v)	Described in section 3.4.3.1. Returns the sum of theexpression e(i, ..., j) evaluated for all combinations of i in u, ..., j in v: e(u[1],...,v[1])+e(u[2],...,v[1])+...+e(u[end],...,v[1])+...+e(u[end],...,v[end]) The type of **sum**(e(i, ..., j) **for** i **in** u, ..., j **in** v) is the same as the type of e(i,...j).
product(A)	Returns the scalar product of all the elements of array expression A. A[1,...,1]*A[2,...,1]*....*A[end,...,1]*A[end,...,end]
product(e(i, ..., j) **for** i **in** u, ..., j **in** v)	Described in section 3.4.3.1. Returns the product of the scalar expression e(i, ..., j) evaluated for all combinations of i in u, ..., j in v: e(u[1],...,v[1])*e(u[2],...,v[1])*...*(u[end],...,v[1])*...*e(u[end],...,v[end]) The type of **product**(e(i, ..., j) **for** i **in** u, ..., j **in** v) is the same as e(i,...j).

symmetric(A)	Returns a matrix where the diagonal elements and the elements above the diagonal are identical to the corresponding elements of matrix A and where the elements below the diagonal are set equal to the elements above the diagonal of A, i.e., B := symmetric(A) -> B[i,j] := A[i,j], if i <= j, B[i,j] := A[j,i], if i > j.
cross(x,y)	Returns the cross product of the 3-vectors x and y, i.e. cross(x,y) = vector([x[2]*y[3]-x[3]*y[2]; x[3]*y[1]-x[1]*y[3]; x[1]*y[2]-x[2]*y[1]]);
skew(x)	Returns the 3 x 3 skew symmetric matrix associated with a 3-vector, i.e., cross(x,y) = skew(x)*y; skew(x) = [0, -x[3], x[2]; x[3], 0, -x[1]; -x[2], x[1], 0];

[Example:

```
Real x[4,1,6];
size(x,1) = 4;
size(x);          // vector with elements 4, 1, 6
size(2*x+x ) = size(x);

Real[3] v1 = fill(1.0, 3);
Real[3,1] m = matrix(v1);
Real[3] v2 = vector(m);

Boolean check[3,4] = fill(true, 3, 4);
```

]

3.4.3.1 Reduction expressions

An expression

```
function-name "(" expression1 for iterators ")"
```

is a reduction-expression. The expressions in the iterators of a reduction-expression shall be vector expressions. They are evaluated once for each reduction-expression, and are evaluated in the scope immediately enclosing the reduction-expression.

For an iterator

```
IDENT in expression2
```

the loop-variable, IDENT, is in scope inside expression1. The loop-variable may hide other variables, as in for-clauses. The result depends on the function-name, and currently the only legal function-names are the built-in operators `array`, `sum`, `product`, `min`, and `max`. For `array`, see section 3.4.4.2. If function-name is `sum`, `product`, `min`, or `max` the result is of the same type as expression1 and is constructed by evaluating expression1 for each value of the loop-variable and computing the `sum`, `product`, `min`, or `max` of the computed elements. For deduction of ranges, see section 3.3.3.1.

Function-name	Restriction on expression1	Result if expression2 is empty
sum	None	zeros(…)
product	Scalar	1
min	Scalar	Modelica.Constants.inf
max	Scalar	-Modelica.Constants.inf

[Example:

```
sum(i for i in 1:10)          // Gives ∑_{i=1}^{10} i = 1+2+...+10=55
// Read it as: compute the sum of i for i in the range 1 to 10.
sum(i^2 for i in {1,3,7,6})   // Gives ∑_{i∈{1 3 7 6}} i^2 = 1+9+49+36=95
{product(j for j in 1:i) for i in 0:4} // Gives {1,1,2,6,24}
max(i^2 for i in {3,7,6})     // Gives 49
```

]

3.4.4 Vector, Matrix and Array Constructors

3.4.4.1 Array Construction

The constructor function **array**(A,B,C,...) constructs an array from its arguments according to the following rules:

- Size matching: All arguments must have the same sizes, i.e., size(A) = size(B) = size(C) = ...
- All arguments must be *type equivalent.* The datatype of the result array is the maximally expanded type of the arguments. The maximally expanded types should be equivalent. Real and Integer subtypes can be mixed resulting in a Real result array where the Integer numbers have been transformed to Real numbers.
- Each application of this constructor function adds a one-sized dimension to the left in the result compared to the dimensions of the argument arrays, i.e., ndims(array(A,B,C)) = ndimes(A) + 1 = ndims(B) + 1, ...
- {A, B, C, ...} is a shorthand notation for array(A, B, C, ...).
- There must be at least one argument *[i.e., array() or {} is not defined]*.

[Examples:

```
{1,2,3} is a 3 vector of type Integer.
{ {11,12,13}, {21,22,23} } is a 2x3 matrix of type Integer
{{{1.0, 2.0, 3.0}}} is a 1x1x3 array of type Real.

Real[3] v = array(1, 2, 3.0);
type Angle = Real(unit="rad");
parameter Angle alpha = 2.0;     // type of alpha is Real.
// array(alpha, 2, 3.0) is a 3 vector of type Real.
Angle[3] a = {1.0, alpha, 4};   // type of a is Real[3].
```

]

3.4.4.2 Array constructor with iterators

An expression

```
"{" expression for iterators "}"
```

or

```
array "(" expression for iterators ")"
```

is an array constructor with iterators. The expressions inside the iterators of an array constructor shall be vector expressions. They are evaluated once for each array constructor, and is evaluated in the scope immediately enclosing the array constructor.

For an iterator

```
IDENT in array_expression
```

the loop-variable, IDENT, is in scope inside expression in the array construction. The loop-variable may hide other variables, as in for-clauses. The loop-variable has the same type as the type of the elements of `array_expression`. For deduction of ranges, see section 3.3.3.1.

Array constructor with one iterator

If only one iterator is used, the result is a vector constructed by evaluating expression for each value of the loop-variable and forming an array of the result.

[Example:

```
array(i for i in 1:10)
// Gives the vector 1:10={1,2,3,...,10}

{r for r in 1.0 : 1.5 : 5.5}
// Gives the vector 1.0:1.5:5.5={1.0, 2.5, 4.0, 5.5}

{i^2 for i in {1,3,7,6}}
// Gives the vector {1, 9, 49, 36}
```

Array constructor with several iterators

The notation with several iterators is a shorthand notation for nested array constructors. The notation can be expanded into the usual form by replacing each ',' by '} `for`' and prepending the array constructor with a '{'.

[Example:

```
Real hilb[:,:]= {(1/(i+j-1) for i in 1:n, j in 1:n};
Real hilb2[:,:]={{(1/(i+j-1) for j in 1:n} for i in 1:n}
```

3.4.4.3 Array Concatenation

The function **cat**(k,A,B,C,...) concatenates arrays A,B,C,... along dimension k according to the following rules:

- Arrays A, B, C, ... must have the same number of dimensions, i.e., ndims(A) = ndims(B) = ...
- Arrays A, B, C, ... must be *type equivalen*t. The datatype of the result array is the maximally expanded type of the arguments. The maximally expanded types should be equivalent. Real and Integer subtypes can be mixed resulting in a Real result array where the Integer numbers have been transformed to Real numbers.
- k has to characterize an existing dimension, i.e., 1 <= k <= ndims(A) = ndims(B) = ndims(C); k shall be an integer number.
- Size matching: Arrays A, B, C, ... must have identical array sizes with the exception of the size of dimension k, i.e., size(A,j) = size(B,j), for 1 <= j <= ndims(A) and j <> k.

[Examples:

```
Real[2,3]  r1  = cat(1, {{1.0, 2.0, 3}}, {{4, 5, 6}});
```

```
    Real[2,6]  r2  = cat(2, r1, 2*r1);
```

]

Concatenation is formally defined according to:

Let R = cat(k,A,B,C,...), and let n = ndims(A) = ndims(B) = ndims(C) = ..., then
size(R,k) = size(A,k) + size(B,k) + size(C,k) + ...
size(R,j) = size(A,j) = size(B,j) = size(C,j) = ..., for 1 <= j <= n and j <> k.
R[i_1, ..., i_k, ..., i_n] = A[i_1, ..., i_k, ..., i_n], for i_k <= size(A,k),
    R[i_1, ..., i_k, ..., i_n] = B[i_1, ..., i_k - size(A,i), ..., i_n], for i_k <= size(A,k) + size(B,k),
     ....
    where 1 <= i_j <= size(R,j) for 1 <= j <= n.

3.4.4.4 Array Concatenation along First and Second Dimensions

For convenience, a special syntax is supported for the concatenation along the first and second dimensions.

- *Concatenation along first dimension:*
  [A; B; C; ...] = cat(1, promote(A,n), promote(B,n), promote(C,n), ...) where
  n = max(2, ndims(A), ndims(B), ndims(C), ...). If necessary, 1-sized dimensions are added to the right of A, B, C before the operation is carried out, in order that the operands have the same number of dimensions which will be at least two.
- *Concatenation along second dimension:*
  [A, B, C, ...] = cat(2, promote(A,n), promote(B,n), promote(C,n), ...) where
  n = max(2, ndims(A), ndims(B), ndims(C), ...). If necessary, 1-sized dimensions are added to the right of A, B, C before the operation is carried out, especially that each operand has at least two dimensions.
- The two forms can be mixed. [...,...] has higher precedence than [...;...], e.g., [a, b; c, d] is parsed as [[a,b]; [c,d]].
- [A] = promote(A,max(2,ndims(A))), i.e., [A] = A, if A has 2 or more dimensions, and it is a matrix with the elements of A, if A is a scalar or a vector.
- There must be at least one argument (i.e. [] is not defined)

[Examples:

```
Real s1, s2, v1[n1], v2[n2], M1[m1,n],
     M2[m2,n], M3[n,m1], M4[n,m2], K1[m1,n,k], K2[m2,n,k];
```

[v1;v2] is a (n1+n2) x 1 matrix
[M1;M2] is a (m1+m2) x n matrix
[M3,M4] is a n x (m1+m2) matrix
[K1;K2] is a (m1+m2) x n x k array
[s1;s2] is a 2 x 1 matrix
[s1,s1] is a 1 x 2 matrix
[s1] is a 1 x 1 matrix
[v1] is a n1 x 1 matrix

```
Real[3] v1 = array(1, 2, 3);
Real[3] v2 = {4, 5, 6};
Real[3,2] m1 = [v1, v2];
Real[3,2] m2 = [v1, [4;5;6]];   // m1 = m2
Real[2,3] m3 = [1, 2, 3; 4, 5, 6];
```

```
   Real[1,3] m4 = [1, 2, 3];
   Real[3,1] m5 = [1; 2; 3];
```
]

3.4.4.5 Vector Construction

Vectors can be constructed with the general array constructor, e.g., `Real[3] v = {1,2,3}`.

The range vector operator or colon operator of *simple-expression* can be used instead of or in combination with this general constructor to construct Real, Integer, Boolean or enumeration type vectors. Semantics of the colon operator:

- j : k is the Integer vector {j, j+1, ..., k}, if j and k are of type Integer.
- j : k is the Real vector {j, j+1.0, ... n}, with n = floor(k-j), if j and/or k are of type Real.
- j : k is a Real, Integer, Boolean, or enumeration type vector with zero elements, if j > k.
- j : d : k is the Integer vector {j, j+d, ..., j+n*d}, with n = (k – j)/d, if j, d, and k are of type Integer.
- j : d : k is the Real vector {j, j+d, ..., j+n*d}, with n = floor((k-j)/d), if j, d, or k are of type Real.
- j : d : k is a Real or Integer vector with zero elements, if d > 0 and j > k or if d < 0 and j < k.
- false : true is the Boolean vector {false, true}.
- j:j is {j} if j is Real, Integer, Boolean, or enumeration type.
- E.ei : E.ej is the enumeration type vector { E.ei, ... E.ej} where E.ej> E.ei, and ei and ej belong to some enumeration type E=enumeration(...ei,...ej,...).

[Examples:

```
   Real v1[5] = 2.7 : 6.8;
   Real v2[5] = {2.7, 3.7, 4.7, 5.7, 6.7};  // = same as v1

   Boolean b1[2] = false:true;

   Colors = enumeration(red,blue,green);

   Colors ec[3] = Colors.red : Colors.green;
```
]

3.4.5 Array access operator

Elements of vector, matrix or array variables are accessed with `[]`. A colon is used to denote all indices of one dimension. A vector expression can be used to pick out selected rows, columns and elements of vectors, matrices, and arrays. The number of dimensions of the expression is reduced by the number of scalar index arguments. The expression **end** may only appear inside array subscripts, and if used in the i:th subscript of an array expression A it is equivalent to size(A,i) provided indices to A are a subtype of Integer. If used inside nested array subscripts it refers to the most closely nested array.

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

[Examples:

- a[:, j] is a vector of the j-th column of a,
- a[j : k] is {[a[j], a[j+1], ... , a[k]}
- a[:,j : k] is [a[:,j], a[:,j+1], ... , a[:,k]],
- v[2:2:8] = v[{2,4,6,8}] .
- v[{j,k}]:={2,3}; // Same as v[j]:=2; v[k]:=3;
- v[{1,1}]:={2,3}; // Same as v[1]:=3;
- A[**end**-1,**end**] is A[size(A,1)-1,size(A,2)]
- A[v[**end**],**end**] is A[v[size(v,1)],size(A,2)] // since the first end is referring to end of v.
- if `x` is a vector, `x[1]` is a scalar, but the slice `x[1:5]` is a vector (a vector-valued or colon index expression causes a vector to be returned).]

[Examples given the declaration x[n, m], v[k], z[i, j, p]:

Expression	# dimensions	Type of value
x[1, 1]	0	Scalar
x[:, 1]	1	n – Vector
x[1, :]	1	m – Vector
v[1:p]	1	p – Vector
x[1:p, :]	2	p x m – Matrix
x[1:1, :]	2	1 x m - "row" matrix
x[{1, 3, 5}, :]	2	3 x m – Matrix
x[: , v]	2	n x k – Matrix
z[: , 3, :]	2	i x p – Matrix
x[scalar([1]), :]	1	m – Vector
x[vector([1]), :]	2	1 x m - "row" matrix

]

3.4.6 Scalar, vector, matrix, and array operator functions

The mathematical operations defined on scalars, vectors, and matrices are the subject of linear algebra.

In all contexts that require an expression which is a subtype of Real, an expression which is a subtype of Integer can also be used; the Integer expression is automatically converted to Real.

The term *numeric class* is used below for a subtype of the Real or Integer type class.

3.4.6.1 Equality and Assignment of type classes

Equality “a=b” and assignment “a:=b” of scalars, vectors, matrices, and arrays is defined element-wise and

require both objects to have the same number of dimensions and corresponding dimension sizes. The operands need to be type equivalent. This is legal for the simple types and all types satisfying the requirements for a record, and is in the latter case applied to each component-element of the records.

Type of a	Type of b	Result of a = b	Operation (j=1:n, k=1:m)
Scalar	Scalar	Scalar	a = b
Vector[n]	Vector[n]	Vector[n]	a[j] = b[j]
Matrix[n, m]	Matrix[n, m]	Matrix[n, m]	a[j, k] = b[j, k]
Array[n, m, …]	Array[n, m, …]	Array[n, m, …]	a[j, k, …] = b[j, k, …]

3.4.6.2 Addition and Subtraction of numeric type classes and concatenation of strings

Addition “a+b” and subtraction “a-b” of numeric scalars, vectors, matrices, and arrays is defined element-wise and require size(a) = size(b) and a numeric type class for a and b. Addition “a+b” of string scalars, vectors, matrices, and arrays is defined as element-wise string concatenation of corresponding elements from a and b, and require size(a) = size(b).

Type of a	Type of b	Result of a +/- b	Operation c := a +/- b (j=1:n, k=1:m)
Scalar	Scalar	Scalar	c := a +/- b
Vector[n]	Vector[n]	Vector[n]	c[j] := a[j] +/- b[j]
Matrix[n, m]	Matrix[n, m]	Matrix[n, m]	c[j, k] := a[j, k] +/- b[j, k]
Array[n, m, …]	Array[n, m, …]	Array[n, m, …]	c [j, k, …] := a[j, k, …] +/- b[j, k, …]

3.4.6.3 Scalar Multiplication of numeric type classes

Scalar multiplication “s*a” or “a*s” with numeric scalar s and numeric scalar, vector, matrix or array a is defined element-wise:

Type of s	Type of a	Type of s* a and a*s	Operation c := s*a or c := a*s (j=1:n, k=1:m)
Scalar	Scalar	Scalar	c := s * a
Scalar	Vector [n]	Vector [n]	c[j] := s* a[j]
Scalar	Matrix [n, m]	Matrix [n, m]	c[j, k] := s* a[j, k]
Scalar	Array[n, m, ...]	Array [n, m, ...]	c[j, k, ...] := s*a[j, k, ...]

3.4.6.4 Matrix Multiplication of numeric type classes

Multiplication “a*b” of numeric vectors and matrices is defined only for the following combinations:

Type of a	Type of b	Type of a* b	Operation c := a*b
Vector [n]	Vector [n]	Scalar	c := sum_k(a[k]*b[k]), k=1:n
Vector [n]	Matrix [n, m]	Vector [m]	c[j] := sum_k(a[k]*b[k, j]), j=1:m, k=1:n
Matrix [n, m]	Vector [m]	Vector [n]	c[j] := sum_k(a[j, k]*b[k])

Matrix [n, m]	Matrix [m, p]	Matrix [n, p]	c[i, j] = sum_k(a[i, k]*b[k, j]), i=1:n, k=1:m, j=1:p

[Example:

```
Real A[3,3], x[3], b[3], v[3];

A*x = b;
x*A = b;                        // same as transpose([x])*A*b
[v]*transpose([v])              // outer product
v*A*v                           // scalar
tranpose([v])*A*v               // vector with one element
```

]

3.4.6.5 Scalar Division of numeric type classes

Division "a/s" of numeric scalars, vectors, matrices, or arrays a and numeric scalars s is defined element-wise. The result is always of real type. In order to get integer division with truncation use the function **div**.

Type of a	Type of s	Result of a / s	Operation c := a / s (j=1:n, k=1:m)
Scalar	Scalar	Scalar	c := a / s
Vector[n]	Scalar	Vector[n]	c[k] := a[k] / s
Matrix[n, m]	Scalar	Matrix[n, m]	c[j, k] := a[j, k] / s
Array[n, m, …]	Scalar	Array[n, m, …]	c[j, k, …] := a[j, k, …] / s

3.4.6.6 Exponentiation of Scalars of numeric type classes

Exponentiation "a^b" is defined as `pow()` in the C language if both "a" and "b" are scalars of a numeric type class.

3.4.6.7 Scalar Exponentiation of Square Matrices of numeric type classes

Exponentiation "a^s" is defined if "a" is a square numeric matrix and "s" is a scalar as a subtype of Integer with s >= 0. The exponentiation is done by repeated multiplication
(e.g. a^3 = a*a*a; a^0 = identity(size(a,1)); assert(size(a,1)==size(a,2),"Matrix must be square"); a^1 = a).

[Non-Integer exponents are forbidden, because this would require to compute the eigenvalues and eigenvectors of "a" and this is no longer an elementary operation].

3.4.6.8 Slice operation

If *a* is an array containing scalar components and *m* is a component of those components, the expression *a.m* is interpreted as a *slice operation*. It returns the array of components {*a*[1].*m*, …}.

If m is also an array component, the slice operation is valid only if size(*a*[1].*m*)=size(*a*[2].*m*)=…

3.4.6.9 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 not be a subtype of Real. *[The reason for this rule is that relations with Real arguments are transformed to state events (see section Events below) and this transformation becomes unnecessarily complicated for the == and <> relational operators (e.g. two crossing functions instead of one crossing function needed, epsilon strategy needed even at event instants). Furthermore, testing on equality of Real variables is questionable on machines where the number length in registers is different to number length in main memory]*.

Relations of the form "v1 rel_op v2", with v1 and v2 variables and rel_op a relational operator are called elementary relations. If either v1 or v2 or both variables are a subtype of Real, the relation is called a Real elementary relation.

3.4.6.10 Boolean operators

The operators, "and" and "or" take expressions of boolean type, which are either scalars or arrays of matching dimensions. The operator "not" takes an expression of boolean type, which is either scalar or an array. The result is the element-wise logical operation. For short-circuit evaluation of "and" and "or" see section 3.4.1.

3.4.6.11 Vectorized call of functions

Functions with one scalar return value can be applied to arrays element-wise, e.g. if A is a vector of reals, then sin(A) is a vector where each element is the result of applying the function sin to the corresponding element in A.

Consider the expression `f(arg1,...,argn)`, an application of the function f to the arguments arg1, ..., argn is defined.

For each passed argument, the type of the argument is checked against the type of the corresponding formal parameter of the function.

1. If the types match, nothing is done.
2. If the types do not match, and a type conversion can be applied, it is applied. Continued with step 1.
3. If the types do not match, and no type conversion is applicable, the passed argument type is checked to see if it is an n-dimensional array of the formal parameter type. If it is not, the function call is invalid. If it is, we call this a foreach argument.
4. For all foreach arguments, the number and sizes of dimensions must match. If they do not match, the function call is invalid. If no foreach argument exists, the function is applied in the normal fashion, and the result has the type specified by the function definition.
5. The result of the function call expression is an n-dimensional array with the same dimension sizes as the foreach arguments. Each element ei,..,j is the result of applying f to arguments constructed from the original arguments in the following way.
   - If the argument is not a foreach argument, it is used as-is.
   - If the argument is a foreach argument, the element at index [i,...,j] is used.

If more than one argument is an array, all of them have to be the same size, and they are traversed in parallel.

[Examples:

```
sin({a, b, c})         = {sin(a), sin(b), sin(c)}     // argument is a vector
sin([a,b,c])           = [sin(a),sin(b),sin(c)]       // argument may be a matrix
atan({a,b,c},{d,e,f}) = {atan(a,d), atan(b,e), atan(c,f)}
```

This works even if the function is declared to take an array as one of its arguments. If pval is defined as a function that takes one argument that is a vector of Reals and returns a Real, then it can be used with an actual argument which is a two-dimensional array (a vector of vectors). The result type in this case will be a vector of Real.

```
pval([1,2;3,4]) = [pval([1,2]); pval([3,4])]

sin([1,2;3,4])  = [sin({1,2}); sin({3,4})]
                = [sin(1), sin(2); sin(3), sin(4)]

function Add
    input  Real e1, e2;
    output Real sum1;
algorithm
   sum1 := e1 + e2;
end Add;
```

Add(1, [1, 2, 3]) adds one to each of the elements of the second argument giving the result [2, 3, 4]. However, it is illegal to write 1 + [1, 2, 3], because the rules for the built-in operators are more restrictive.]

3.4.6.12 Empty Arrays

Arrays may have dimension sizes of 0. E.g.

```
Real x[0];                      // an empty vector
Real A[0, 3], B[5, 0], C[0, 0]; // empty matrices
```

- Empty matrices can be constructed with the fill function. E.g.

```
Real    A[:,:]   = fill(0.0, 0, 1);        // a Real 0 x 1 matrix
Boolean B[:, :, :] = fill(false, 0, 1, 0); // a Boolean 0 x 1 x 0 matrix
```

- It is not possible to access an element of an empty matrix, e.g. v[j,k] is wrong if "v=[]" because the assertion fails that the index must be bigger than one.
- Size-requirements of operations, such as +, -, have also to be fulfilled if a dimension is zero. E.g.

```
Real[3,0] A, B;
Real[0,0] C;
A + B   // fine, result is an empty matrix
A + C   // error, sizes do not agree
```

- Multiplication of two empty matrices results in a zero matrix if the result matrix has no zero dimension sizes, i.e.,

```
Real[0,m]*Real[m,n] = Real[0,n]  (empty matrix)
Real[m,n]*Real[n,0] = Real[m,0]  (empty matrix)
Real[m,0]*Real[0,n] = zeros(m,n) (non-empty matrix, with zero elements).
```

[Example:

```
Real u[p], x[n], y[q], A[n,n], B[n,p], C[q,n], D[q,p];

der(x) = A*x + B*u
```

```
        y = C*x + D*u
```

*Assume n=0, p>0, q>0: Results in "y = D*u"*

]

3.4.7 If-expression

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 and give the type of the if-expression. If-expressions with **elseif** are defined by replacing **elseif** by **else if**.*[Note: elseif is added for symmetry with if-clauses.]* For short-circuit evaluation see section 3.4.1.

[Example:

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

]

3.4.8 Functions

Function classes and record constructors can be called as described in this section. It also possible to declare components of function type, see section 3.2.13.

3.4.8.1 Formal input parameters of functions

A function application, see section 2.2.7, has optional positional arguments followed by zero, one or more named arguments, such as

```
f(3.5, 5.76, arg3=5, arg6=8.3);
```

The interpretation of a function application is as follows: First, a list of unfilled slots is created for all formal input parameters. If there are N positional arguments, they are placed in the first N slots, where the order of the parameters is given by the order of the component declarations in the function definition. Next, for each named argument "identifier = expression", the identifier is used to determine the corresponding slot. This slot shall be not filled *[otherwise an error occurs]* and the value of the argument is placed in the slot, filling it. When all arguments have been processed, the slots that are still unfilled are filled with the corresponding default value of the function definition. There shall be no remaining unfilled slots *[otherwise an error occurs]* and the list of filled slots is used as the argument list for the call.

The type of each argument must agree with the type of the corresponding parameter, except where the standard type coercions can be used to make the types agree. (See also section 3.4.6.10 on applying scalar functions to arrays.)

[Example.

Suppose a function RealToString is defined as follows to convert a Real number to a String:

```
function RealToString
  input  Real number;
```

```
  input  Real precision = 6 "number of significantdigits";
  input  Real length    = 0 "minimum length of field";
  output String string "number as string";
    ...
end RealToString;
```

Then the following applications are equivalent:

```
  RealToString(2.0);
  RealToString(2.0, 6, 0);
  RealToString(2.0, 6);
  RealToString(2.0, precision=6);
  RealToString(2.0, length=0);
  RealToString(2.0, 6, precision=6);  // error: slot is used twice
```

]

3.4.8.2 Formal output parameters of functions

A function may have more than one output component, corresponding to multiple return values. The only way to call a function returning more than one result is to make the function call the right hand side of an equation or assignment. In these cases, the left hand side of the equation or assignment shall contain a list of component references within parentheses:

```
(out1, out2, out3) = f(...);
```

The component references are associated with the output components according to their position in the list. Thus output component i is set equal to, or assigned to, component reference i in the list, where the order of the output components is given by the order of the component declarations in the function definition. The type of each component reference in the list must agree with the type of the corresponding output component.

A function application may be used as expression whose value and type is given by the value and type of the first output component, if exactly one return result is provided.

It is possible to omit left hand side component references and/or truncate the left hand side list in order to discard outputs from a function call. *[The operator isPresent(out2) can be used inside the function to determine if the formal output out2 needs to be computed, see section 3.4.2.]*

[Example:

Function "eigen" to compute eigenvalues and optionally eigenvectors may be called in the following ways:

```
        ev = eigen(A);                  // calculate eigenvalues
        x  = isStable(eigen(A));        // used in an expression
  (ev, vr) = eigen(A)                   // calculate eigenvectors
(ev,vr,vl) = eigen(A)                   // and also left eigenvectors
(ev,,vl)   = eigen(A)                   // no right eigenvectors
```

The function may be defined as:

```
  function eigen "calculate eigenvalues and optionally eigenvectors"
    parameter Integer n = size(A,1);
    input     Real    A[:, size(A,1)];
    output    Real    eigenValues[n,2];
    output    Real    rightEigenVectors[n,n];
    output    Real    leftEigenVectors [n,n];
```

```
  algorithm
    // compute eigenvalues
    if isPresent(rightEigenVectors) then
       // compute right eigenvectors
    end if;
    if isPresent(leftEigenVectors) then
       // compute left eigenvectors
    end if;
  end eigen;
```

]

The only permissible use of an expression in the form of a list of expressions in parentheses, is when it is used as the left hand side of an equation or assignment where the right hand side is an application of a function.

[Example. The following are illegal:

```
(x+1, 3.0, z/y) = f(1.0, 2.0);     // Not a list of component references.
(x, y, z) + (u, v, w)              // Not LHS of suitable eqn/assignment.
```

]

3.4.8.3 Record constructor

Whenever a record is defined, a record constructor function with the same name and in the same scope as the record class is implicitly defined according to the following rules:

- The declaration of the record is partially instantiated including inheritance, modifications, redeclarations, and expansion of all names referring to declarations outside of the scope of the record to their fully qualified names *[in order to remove potentially conflicting import statements in the record constructor function due to flattening the inheritance tree]*.
- All record elements *[i.e., components and local class definitions]* of the partially instantiated record declaration are used as declarations in the record constructor function with the following exceptions: (1) Component declarations which do not allow a modification *[such as "**constant** Real c=1" or "**final parameter** Real"]* are declared as protected components in the record constructor function. (2) Prefixes (**constant**, **parameter**, **final**, **discrete**, **input**, **output**, ...) of the remaining record components are removed. (3) The prefix "**input**" is added to the **public** components of the record constructor function.
- An instance of the record is declared as output parameter *[using a name, not appearing in the record]* together with a modification. In the modification, all input parameters are used to set the corresponding record variables.

[This allows to construct an instance of a record, with an optional modification, at all places where a function call is allowed. Examples:

```
record Complex "Complex number"
  Real re "real part";
  Real im "imaginary part";
end Complex;

function add
  input  Complex u, v;
  output Complex w(re=u.re + v.re, im=u.im+v.re);
end add;
```

```
  Complex c1, c2;
equation
  c2 = add(c1, Complex(sin(time), cos(time));
```

In the following example, a convenient data sheet library of components is built up:

```
package Motors
  record MotorData "Data sheet of a motor"
     parameter Real inertia;
     parameter Real nominalTorque;
     parameter Real maxTorque;
     parameter Real maxSpeed;
  end MotorData;

  model Motor "Motor model"  // using the generic MotorData
     MotorData data;
       ...
  equation
       ...
  end Motor;

  record MotorI123 = MotorData(  // data of a specific motor
     inertia       = 0.001,
     nominalTorque = 10,
     maxTorque     = 20,
     maxSpeed      = 3600) "Data sheet of motor I123";

  record MotorI145 = MotorData(  // data of another specific motor
     inertia       = 0.0015,
     nominalTorque = 15,
     maxTorque     = 22,
     maxSpeed      = 3600) "Data sheet of motor I145";
end Motors

model Robot
   import Motors.*;
   Motor motor1(data = MotorI123());   // just refer to data sheet
   Motor motor2(data = MotorI123(inertia=0.0012));
     // data can still be modified (if no final declaration in record)
   Motor motor3(data = MotorI145());
     ...
end Robot;
```

Example showing most of the situations, which may occur for the implicit record constructor function creation. With the following record definitions

```
package Demo;
  record Record1;
     parameter Real r0 = 0;
```

```
  end Record1;

  record Record2
     import Modelica.Math.*;
     extends Record1;
          constant  Real    c1 = 2.0;
          constant  Real    c2;
          parameter Integer n1 = 5;
          parameter Integer n2;
          parameter Real    r1 "comment";
          parameter Real    r2 = sin(c1);
    final parameter Real    r3 = cos(r2);
                    Real    r4;
                    Real    r5 = 5.0;
                    Real    r6[n1];
                    Real    r7[n2];
  end Record2;
end Demo;
```

the following record constructor functions are implicitly defined

```
package Demo;
  function Record1
    input  Real r0 = 0;
    output Record1 'result'(r0 = r0);
  end Record1;

  function Record2
    input  Real    r0 = 0;
    input  Real    c2;
    input  Integer n1 := 5;
    input  Integer n2;
    input  Real    r1 "comment";  // the comment also copied from record
    input  Real    r2 := Modelica.Math.sin(c1);
    input  Real    r4;
    input  Real    r5 = 5.0;
    input  Real    r6[n1];
    input  Real    r7[n2];
    output Record2 'result'(r0=r0,c2=c2,n1=n1,n2=n2,r1=r1,r2=r2,
                            r4=r4,r5=r5,r6=r6,r7=r7);

  protected
    constant        Real c1 = 2.0; // referenced from r2
    final parameter Real r3 = Modelica.Math.cos(r2);

  end Record2;
end Demo;
```

and can be applied in the following way

```
Demo.Record2 r1 = Demo.Record2(r0=1, c2=2, n1=2, n2=3, r1=1, r2=2,
```

```
                                    r4=5, r5=5, r6={1,2}, r7={1,2,3});
Demo.Record2 r2 = Demo.Record2(1,2,2,3,1,2,5,5,{1,2},{1,2,3});
parameter Demo.Record2 r3 = Demo.Record2(c2=2, n2=1, r1=1, r4=4,
                                         r6=1:5, r7={1});
```

The above example is only used to show the different variants appearing with prefixes, but it is not very meaningful, because it is simpler to just use a direct modifier.

]

3.4.8.4 Type conversion constructor

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

String(E.enumvalue) gives the string representation of the enumeration value.

[Example: String(E.Small) gives "Small".]

3.4.9 Variability of 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, pre** and **analysisType** a function or operator with constant subexpressions as argument (and no parameters defined in the function) is a constant expression.

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.
- The function **analysisType**() is parameter expression.

Discrete-time expressions are:

- Parameter expressions.
- Discrete-time variables, i.e. Integer, Boolean, String variables and enumeration variables, as well as Real variables assigned in when-clauses
- Function calls where all input arguments of the function are **discrete-time** expressions.
- Expressions where all the subexpressions are **discrete-time** expressions.
- Expressions in the body of a **when** clause.
- Unless inside **noEvent**: Ordered relations (>,<,>=,<=) and the functions **ceil**, **floor**, **div**, **mod**, **rem**, **abs**, **sign**. These will generate events if they have continuous-time subexpressions. *[In other words, relations inside **noEvent**(), such as **noEvent**(x > 1), are continuous-time expressions]*.

- The functions **pre**, **edge**, and **change** result in discrete-time expressions.
- Expressions in functions behave as though they were **discrete-time** expressions.

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

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

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

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

Inside an if-expression, if-clause or for-clause, that is controlled by a continuous-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 are no continuous-time equations for discrete variables, 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 and
                      // noEvent(x > 1) is a continuous-time expression.
end Test;
```

]

3.5 Events and Synchronization

The integration is halted and an event occurs whenever a Real elementary relation, e.g. “x > 2”, changes its value. The value of a relation can only be changed at event instants *[in other words, Real elementary relations*

induce state or time events]. The relation which triggered an event changes its value when evaluated literally before the model is processed at the event instant *[in other words, a root finding mechanism is needed which determines a small time interval in which the relation changes its value; the event occurs at the **right** side of this interval]*. Relations in the body of a when-clause are always taken literally. During continuous integration a Real elementary relation has the *constant* value of the relation from the last event instant.

[Example:

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

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

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

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

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

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

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

[Example:

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

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

*The **smooth** function is useful, if e.g. the modeller can guarantee that the used if-clauses fulfill at least the continuity requirement of integrators. In this case the simulation speed is improved, since no state event iterations occur during integration. The **noEvent** function is used to guard against "outside domain" errors, e.g. y = **if noEvent**(x >= 0) **then** sqrt(x) **else** 0.]*

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

[Example:

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

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

Modelica is based on the synchronous data flow principle which is 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).

[These rules 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 behaviour. Furthermore, the continuous and the discrete parts of a model are always automatically "synchronized". Example:

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

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

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

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

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

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

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

```
       Boolean fastSample, slowSample;
```

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

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

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

3.6 Predefined types

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

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

type Integer
  IntegerType value;                   // Accessed without dot-notation
```

```
  parameter StringType  quantity    = "";
  parameter IntegerType min=-Inf, max=+Inf;
  parameter IntegerType start = 0;     // Initial value
  parameter BooleanType fixed = true,  // default for parameter/constant;
                              = false; // default for other variables
equation
  assert(value >= min and value <= max, "Variable value out of limit");
end Integer;

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

type String
  StringType value;                    // Accessed without dot-notation
  parameter StringType  quantity    = "";
  parameter StringType start = "";     // Initial value
end String;

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

For each enumeration

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

a new simple type is conceptually defined as

```
type E
  EnumType value;                   // Accessed without dot-notation
  parameter StringType  quantity    = "";
  parameter EnumType min=e1, max=en;
  parameter EnumType start = e1;     // Initial value
  parameter BooleanType fixed = true,  // default for parameter/constant;
                              = false; // default for other variables
  constant EnumType e1=...;
  ...
  constant EnumType en=...;
```

```
equation
  assert(value >= min and value <= max, "Variable value out of limit");
end E;
```

The attributes “start” and “fixed” define the initial conditions for a variable for analysisType = "static". “fixed=false” means an initial guess, i.e., value may be changed by static analyzer. “fixed=true” means a required value. Before other analysisTypes (such as "dynamic") are performed, the analysisType "static" has to be carried out first. The resulting consistent set of values for ALL model variables is used as initial values for the analysis to be performed.

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

3.7 Built-in variable time

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

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

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

[Example:

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

]

4 Mathematical description of Hybrid DAEs

In this section, the mapping of a Modelica model into an appropriate mathematical description form is discussed.

In a first step, a Modelica translator transforms a hierarchical Modelica model into a "flat" set of Modelica statements, consisting of the equation and algorithm sections of all used components by:

- expanding all class definitions (flattening the inheritance tree) and adding the equations and assignment statements of the expanded classes for every instance of the model
- replacing all connect-statements by the corresponding equations of the connection set (see 3.3.8.1).
- mapping all algorithm sections to equation sets.
- mapping all when clauses to equation sets (see 3.3.4).

As a result of this transformation process, a *set of equations* is obtained consisting of *differential, algebraic* and *discrete equations* of the following form ($\mathbf{v} := [\dot{\mathbf{x}};\ \mathbf{x};\ \mathbf{y};\ t;\ \mathbf{m};\ \mathbf{pre}(\mathbf{m});\ \mathbf{p}]$):

$$\text{(1a)} \quad \mathbf{c} := \mathbf{f}_c(relation(\mathbf{v}))$$

$$\text{(1b)} \quad \mathbf{m} := \mathbf{f}_m(\mathbf{v}, \mathbf{c})$$

$$\text{(1c)} \quad \mathbf{0} = \mathbf{f}_x(\mathbf{v}, \mathbf{c})$$

where

p	Modelica variables declared as *parameter* or *constant*, i.e., variables without any time-dependency.
t	Modelica variable *time*, the independent (real) variable.
x(t)	Modelica variables of type *Real*, appearing differentiated.
m(t_e)	Modelica variables of type *discrete Real, Boolean, Integer* which are unknown. These variables change their value only at event instants t_e. **pre**(**m**) are the values of **m** immediately before the current event occurred.
y(t)	Modelica variables of type *Real* which do not fall into any other category (= algebraic variables).
c(t_e)	The conditions of all **if**- expressions generated including **when**-clauses after conversion, see 3.3.4).
relation(**v**)	A relation containing variables v_i, e.g. $v_1 > v_2$, $v_3 >= 0$.

For simplicity, the special cases of the **noEvent**() operator and of the **reinit**() operator are not contained in the equations above and are not discussed below.

The generated set of equations is used for simulation and other analysis activities. Simulation means that an initial value problem is solved, i.e., initial values have to be provided for the states **x,** section 3.3.9. The equations define a DAE (Differential Algebraic Equations) which may have discontinuities, a variable structure and/or which are controlled by a discrete-event system. Such types of systems are called *hybrid DAEs*. Simulation is performed in the following way:

1. The DAE (1c) is solved by a numerical integration method. In this phase the conditions **c** of the if- and when-clauses, as well as the discrete variables **m** are kept constant. Therefore, (1c) is a continuous function of continuous variables and the most basic requirement of numerical integrators is fulfilled.
2. During integration, all relations from (1a) are monitored. If one of the relations changes its value an event is triggered, i.e., the exact time instant of the change is determined and the integration is halted. As discussed in section 3.5, relations which depend only on time are usually treated in a special way, because this allows to determine the time instant of the next event in advance.
3. At an event instant, (1) is a mixed set of algebraic equations which is solved for the Real, Boolean and Integer unknowns.
4. After an event is processed, the integration is restarted with 1.

Note, that both the values of the conditions **c** as well as the values of **m** (all *discrete Real, Boolean* and *Integer* variables) are only changed at an event instant and that these variables remain constant during continuous integration. At every event instant, new values of the discrete variables **m** and of new initial values for the states **x** are determined. The change of discrete variables may characterize a new structure of a DAE where elements of the state vector **x** are *disabled*. In other words, the number of state variables, algebraic variables and residue equations of a DAE may change at event instants by disabling the appropriate part of the DAE. For clarity of the equations, this is not explicitly shown by an additional index in (1).

At an event instant, including the initial event, the model equations are reinitialized according to the following iteration procedure:

```
known  variables: x, t, p
unkown variables: dx/dt, y, m, pre(m), c

// pre(m) = value of m before event occured
loop
  solve (1) for the unknowns, with pre(m) fixed
  if m == pre(m) then break
  pre(m) := m
end loop
```

Solving (1) for the unknowns is non-trivial, because this set of equations contains not only Real, but also Boolean and Integer unknowns. Usually, in a first step these equations are sorted and in many cases the Boolean and Integer unknowns can be just computed by a forward evaluation sequence. In some cases, there remain systems of equations (e.g. for ideal diodes, Coulomb friction elements) and specialized algorithms have to be used to solve them.

Due to the construction of the equations by "flattening" a Modelica model, the hybrid DAE (1) contains a huge number of sparse equations. Therefore, direct simulation of (1) requires sparse matrix methods. However, solving this initial set of equations directly with a numerical method is both unreliable and inefficient. One reason is that many Modelica models, like the mechanical ones, have a DAE index of 2 or 3, i.e., the overall number of states of the model is less than the sum of the states of the sub-components. In such a case, every direct numerical method has the difficulty that the numerical condition becomes worse, if the integrator step size is reduced and that a step size of zero leads to a singularity. Another problem is the handling of idealized

elements, such as ideal diodes or Coulomb friction. These elements lead to *mixed* systems of equations having both *Real* and *Boolean unknowns*. Specialized algorithms are needed to solve such systems.

To summarize, symbolic transformation techniques are needed to transform (1) in a set of equations which can be numerically solved reliably. Most important, the algorithm of Pantelides should to be applied to differentiate certain parts of the equations in order to reduce the index. Note, that also *explicit integration* methods, such as Runge-Kutta algorithms, can be used to solve (1c), after the index of (1c) has been reduced by the Pantelides algorithm: During continuous integration, the integrator provides **x** and t. Then, (1c) is a linear or nonlinear system of equations to compute the algebraic variables **y** and the state derivatives d**x**/dt and the model returns d**x**/dt to the integrator by solving these systems of equations. Often, (1c) is just a linear system of equations in these unknowns, so that the solution is straightforward. This procedure is especially useful for *real-time* simulation where usually explicit one-step methods are used.

5 Unit expressions

Unless otherwise stated, the syntax and semantics of unit expressions in Modelica are conform with the international standards ISO 31/0-1992 "General principles concerning quantities, units and symbols" and ISO 1000-1992 "SI units and recommendations for the use of their multiples and of certain other units". Unfortunately, neither these two standards nor other existing or emerging ISO standards define a formal syntax for unit expressions. There are recommendations and Modelica exploits them.

Examples for the syntax of unit expressions used in Modelica: "N.m", "kg.m/s2", "kg.m.s-2" "1/rad", "mm/s".

5.1 The Syntax of unit expressions

```
unit_expression:
   unit_numerator [ "/" unit_denominator ]

unit_numerator:
   "1" | unit_factors |  "(" unit_expression ")"

unit_denominator:
   unit_factor |  "(" unit_expression ")"
```

The unit of measure of a dimension free quantity is denoted by "1". The ISO standard does not define any precedence between multiplications and divisions. The ISO recommendation is to have at most one division, where the expression to the right of "/" either contains no multiplications or is enclosed within parentheses. It is also possible to use negative exponents, for example, "J/(kg.K)" may be written as "J.kg-1.K-1".

```
unit_factors:
   unit_factor [ unit_mulop  unit_factors ]

unit_mulop:
   "."
```

The ISO standard allows that a multiplication operator symbol is left out. However, Modelica enforces the ISO recommendation that each multiplication operator is explicitly written out in formal specifications. For example, Modelica does not support "Nm" for newtonmeter, but requires it to written as "N.m".

The preferred ISO symbol for the multiplication operator is a "dot" a bit above the base line: "·". Modelica supports the ISO alternative ".", which is an ordinary "dot" on the base line.

```
unit_factor:
   unit_operand [ unit_exponent ]

unit_exponent:
   [ "+" | "-" ] integer
```

The ISO standard does not define any operator symbol for exponentiation. A unit_factor consists of a unit_operand possibly suffixed by a possibly signed integer number, which is interpreted as an exponent. There must be no spacing between the unit_operand and a possible unit_exponent.

```
unit_operand:
   unit_symbol | unit_prefix unit_symbol

unit_prefix:
  Y | Z | E | P | T | G | M | k | h | da | d | c | m | u | p | f | a | z |
y
```

A unit_symbol is a string of letters. A basic support of units in Modelica should know the basic and derived units of the SI system. It is possible to support user defined unit symbols. In the base version Greek letters is not supported, but full names must then be written, for example "Ohm".

A unit_operand should first be interpreted as a unit_symbol and only if not successful the second alternative assuming a prefixed operand should be exploited. There must be no spacing between the unit_symbol and a possible unit_prefix. The value of the prefixes are according to the ISO standard. The letter "u" is used as a symbol for the prefix micro.

5.2 Examples

- The unit expression "m" means meter and not milli (10^{-3}), since prefixes cannot be used in isolation. For millimeter use "mm" and for squaremeter, m^2, write "m2".
- The expression "mm2" means $mm^2 = (10^{-3}m)^2 = 10^{-6}m^2$. Note that exponentiation includes the prefix.

  The unit expression "T" means Tesla, but note that the letter "T" is also the symbol for the prefix tera which has a multiplier value of 10^{12}.

6 External function interface

6.1 Overview

Here, the word function is used to refer to an arbitrary external routine, whether or not the routine has a return value or returns its result via output parameters (or both). The Modelica external function call interface provides the following:

- Support for external functions written in C and FORTRAN 77. Other languages, e.g. C++ and Fortran 90, may be supported in the future.
- Mapping of argument types from Modelica to the target language and back.
- Natural type conversion rules in the sense that there is a mapping from Modelica to standard libraries of the target language.
- Handling arbitrary parameter order for the external function.
- Passing arrays to and from external functions where the dimension sizes are passed as explicit integer parameters.
- Handling of external function parameters which are used both for input and output.

The format of an external function declaration is as follows.

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

Components in the public part of an external function declaration shall be declared either as **input** or **output**. *[This is just as for any other function. The components in the protected part allows local variables for temporary storage to be declared.]*

The *language-specification* must currently be one of `"C"` or `"FORTRAN 77"`. Unless the external language is specified, it is assumed to be C.

The *external-function-call* specification allows functions whose prototypes do not match the default assumptions as defined below to be called. It also gives the name used to call the external function. If the external call is not given explicitly, this name is assumed to be the same as the Modelica name.

The only permissible kinds of expressions in the argument list are identifiers, scalar constants, and the function `size` applied to an array and a constant dimension number. The annotations are used to pass additional information to the compiler when necessary. Currently, the only supported annotation is `arrayLayout`, which can be either "`rowMajor`" or "`columnMajor`".

6.2 Argument type mapping

The arguments of the external function are declared in the same order as in the Modelica declaration, unless specified otherwise in an explicit external function call. Protected variables (i.e. temporaries) are passed in the same way as outputs, whereas constants and size-expression are passed as inputs.

6.2.1 Simple types

Arguments of **simple** types are by default mapped as follows for C:

Modelica	C	
	Input	Output
`Real`	`double`	`double *`
`Integer`	`int`	`int *`
`Boolean`	`int`	`int *`
`String`	`const char *`	`const char **`
`Enumeration type`	`int`	`int *`

An exception is made when the argument is of the form `size(…, …)`. In this case the corresponding C-type is `size_t`.

Strings are NUL-terminated (i.e., terminated by '\0') to facilitate calling of C functions. When returning a non-literal string, the memory for this string can be allocated with function "ModelicaAllocateString" (see section 6.6) *[It is not suitable to use malloc, because a Modelica simulation environment may have its own allocation scheme, e.g., a special stack for local variables of a function]*. After return of the external function, the Modelica environment is responsible for the memory allocated with ModelicaAllocateString (e.g., to free this memory, when appropriate). It is not allowed to access memory that was allocated with ModelicaAllocateString in a previous call of this external function.

Arguments of **simple** types are by default mapped as follows for FORTRAN 77:

Modelica	FORTRAN 77	
	Input	Output
`Real`	`DOUBLE PRECISION`	`DOUBLE PRECISION`
`Integer`	`INTEGER`	`INTEGER`
`Boolean`	`LOGICAL`	`LOGICAL`
`Enumeration type`	`INTEGER`	`INTEGER`

Passing strings to FORTRAN 77 subroutines/functions is currently not supported.

Enumeration types used as arguments are mapped to type `int` when calling an external C function, and to type `INTEGER` when calling an external FORTRAN function. The i:th enumeration literal is mapped to integer value i, starting at one.

Return values are mapped to enumeration types analogously: integer value 1 is mapped to the first enumeration literal, 2 to the second, etc. Returning a value which does not map to an existing enumeration literal for the

specified enumeration type is an error.

6.2.2 Arrays

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

Arrays are by default stored in row-major order when calling C functions and in column-major order when calling FORTRAN 77 functions. These defaults can be overridden by the array layout annotation. See the example below.

The table below shows the mapping of an array argument in the absence of an explicit external function call when calling a C function. The type *T* is allowed to be any of the simple types which can be passed to C as defined in section 6.2.1 or a record type as defined in section 6.2.3 and it is mapped to the type *T'* as defined in these sections.

Modelica	C
	Input and Output
$T[dim_1]$	T' *, size_t dim_1
$T[dim_1,dim_2]$	T' *, size_t dim_1, size_t dim_2
$T[dim_1, \ldots, dim_n]$	T' *, size_t dim_1, …, size_t dim_n

The method used to pass array arguments to FORTRAN 77 functions in the absence of an explicit external function call is similar to the one defined above for C: first the address of the array, then the dimension sizes as integers. See the table below. The type *T* is allowed to be any of the simple types which can be passed to FORTRAN 77 as defined in section 6.2.1 and it is mapped to the type *T'* as defined in that section.

Modelica	FORTRAN 77
	Input and Output
$T[dim_1]$	T', INTEGER dim_1
$T[dim_1,dim_2]$	T', INTEGER dim_1, INTEGER dim_2
$T[dim_1, \ldots, dim_n]$	T', INTEGER dim_1, …, INTEGER dim_n

[The following two examples illustrate the default mapping of array arguments to external C and FORTRAN 77 functions.

```
function foo
   input   Real     a[:,:,:];
   output  Real     x;
   external;
end foo;
```

The corresponding C prototype is as follows:

```
double foo(double *, size_t, size_t, size_t);
```

If the external function is written in FORTRAN 77, i.e.:

```
function foo
   input   Real     a[:,:,:];
   output  Real     x;
   external "FORTRAN 77";
end foo;
```

the default assumptions correspond to a FORTRAN 77 function defined as follows:

```
FUNCTION foo(a, d1, d2, d3)
   DOUBLE PRECISION(d1,d2,d3) a
   INTEGER                    d1
   INTEGER                    d2
   INTEGER                    d3
   DOUBLE PRECISION           foo
   ...
END
```

]

When an explicit call to the external function is present, the array and the sizes of its dimensions must be passed explicitly.

[This example shows how to arrays can be passed explicitly to an external FORTRAN 77 function when the default assumptions are unsuitable.

```
function foo
   input   Real     x[:];
   input   Real     y[size(x,1),:];
   input   Integer  i;
   output  Real     u1[size(y,1)];
   output  Integer  u2[size(y,2)];
   external "FORTRAN 77" myfoo(x, y, size(x,1), size(y,2),
                               u1, i, u2);
end foo;
```

The corresponding FORTRAN 77 subroutine would be declared as follows:

```
SUBROUTINE myfoo(x, y, n, m, u1, i, u2)
   DOUBLE PRECISION(n)   x
   DOUBLE PRECISION(n,m) y
   INTEGER               n
   INTEGER               m
   DOUBLE PRECISION(n)   u1
   INTEGER               i
   DOUBLE PRECISION(m)   u2
   ...
END
```

This example shows how to pass an array in column major order to a C function.

```
function fie
   input  Real[:,:] a;
   output Real b;
   external;
   annotation(arrayLayout = "columnMajor");
end fie;
```

This corresponds to the following C-prototype:

```
    double fie(double *, size_t, size_t);
```
]

6.2.3 Records

Mapping of record types is only supported for C. A Modelica record class that contains simple types, other record elements, or arrays with fixed dimensions thereof, is mapped as follows:

- The record class is represented by a struct in C.
- Each element of the Modelica record is mapped to its corresponding C representation.
  The elements of the Modelica record class are declared in the same order in the C struct.
- Arrays are mapped to the corresponding C array, taking the default array layout or any explicit `arrayLayout`-directive into consideration.
- Records are passed by reference (i.e. a pointer to the record is being passed).

For example,

```
record R                                    struct R {
  Real x;                                     double x;
  Integer y[10];       is mapped to           int    y[10];
  Real z;                                     double z;
end R;                                      };
```

6.3 Return type mapping

If there is a single output parameter and no explicit call of the external function, or if there is an explicit external call in the form of an equation, in which case the LHS must be one of the output parameters, the external routine is assumed to be a value-returning function. Mapping of the return type of functions is performed as indicated in the table below. Storage for arrays as return values is allocated by the calling routine, so the dimensions of the returned array are fixed at call time. Otherwise the external function is assumed not to return anything; i.e., it is really a procedure or, in C, a `void`-function. *[In this case, argument type mapping according to section 6.2 is performed in the absence of any explicit external function call.]*

Return types are by default mapped as follows for C and FORTRAN 77:

Modelica	C	FORTRAN 77
`Real`	`double`	`DOUBLE PRECISION`
`Integer`	`int`	`INTEGER`
`Boolean`	`int`	`LOGICAL`
`String`	`const char*`	Not allowed.
T`[`dim_1`, …, `dim_n`]`	*T'* `*`	*T'*
Enumeration type	`int`	`INTEGER`
Record	See section 6.2.3.	Not allowed.

The element type *T* of an array can be any simple type as defined in section 6.2.1 or, for C, a record type as defined in section 6.2.3. The element type *T* is mapped to the type *T'* as defined in these sections.

6.4 Aliasing

Any potential aliasing in the external function is the responsibility of the tool and not the user. An external function is not allowed to internally change the inputs (even if they are restored before the end of the function).
[Example:

```
function foo
  input Real x;
  input Real y;
  output Real z:=x;
  external "FORTRAN 77" myfoo(x,y,z);
end foo;
```

The following Modelica function:

```
function f
  input  Real a;
  output Real b;
algorithm
  b:=foo(a,a);
  b:=foo(b,2*b);
end f;
```

can on most systems be transformed into the following C function

```
double f(double a) {
  extern void myfoo_(double*,double*,double*);
  double b,temp1,temp2;
  myfoo_(&a,&a,&b);
  temp1=2*b;
  temp2=b;
  myfoo_(&b,&temp1,&temp2);
  return temp2;
}
```

The reason for not allowing the external function to change the inputs is to ensure that inputs can be stored in static memory and to avoid superfluous copying (especially of matrices). If the routine does not satisfy the requirements the interface must copy the input argument to a temporary. This is rare but occurs e.g. in dormlq in some Lapack implementations. In those special cases the writer of the external interface have to copy the input to a temporary. If the first input was changed internally in myfoo the designer of the interface would have to change the interface function "foo" to:

```
function foo
  input Real x;
  protected Real xtemp:=x; // Temporary used because myfoo changes its input
  public input Real y;
  output Real z;
  external "FORTRAN 77" myfoo(xtemp,y,z);
end foo;
```

Note that we discuss input arguments for Fortran-routines even though Fortran 77 does not formally have input arguments and forbid aliasing between any pair of arguments to a function (section 15.9.3.6 of X3J3/90.4). For the few (if any) Fortran 77 compilers that strictly follow the standard and are unable to handle aliasing between input variables the tool must transform the first call of foo into

```
temp1=a; /* Temporary to avoid aliasing */
```

```
myfoo_(&a,&temp1,&b);
```

The use of the function foo in Modelica is uninfluenced by these considerations.]

6.5 Examples

6.5.1 Input parameters, function value

[Here all parameters to the external function are input parameters. One function value is returned. If the external language is not specified, the default is "C", as below.

```
function foo
   input  Real     x;
   input  Integer  y;
   output Real     w;
   external;
end foo;
```

This corresponds to the following C-prototype:

```
double foo(double, int);
```

Example call in Modelica:

```
z = foo(2.4, 3);
```

Translated call in C:

```
z = foo(2.4, 3);
```

6.5.2 Arbitrary placement of output parameters, no external function value

In the following example, the external function call is given explicitly which allows passing the arguments in a different order than in the Modelica version.

```
function foo
   input  Real     x;
   input  Integer  y;
   output Real     u1;
   output Integer  u2;
   external "C"  myfoo(x, u1, y, u2);
end foo;
```

This corresponds to the following C-prototype:

```
void myfoo(double, double *, int, int *);
```

Example call in Modelica:

```
(z1,i2) = foo(2.4, 3);
```

Translated call in C:

```
myfoo(2.4, &z1, 3, &i2);
```

6.5.3 External function with both function value and output variable

The following external function returns two results: one function value and one output parameter value. Both are mapped to Modelica output parameters.

```
function foo
   input  Real     x;
   input  Integer  y;
```

```
    output Real     funcvalue;
    output Integer  out1;
    external "C" funcvalue = myfoo(x, y, out1);
  end foo;
```

This corresponds to the following C-prototype:

```
  double myfoo(double, int, int *);
```

Example call in Modelica:

```
  (z1,i2) = foo(2.4, 3);
```

Translated call in C:

```
  z1 = myfoo(2.4, 3, &i2);
```

]

6.6 Utility functions

The following utility functions can be called in external Modelica functions written in C. These functions are defined in file **ModelicaUtilities.h**:

ModelicaMessage	*void ModelicaMessage(const char* string)* Output the message string (no format control).
ModelicaFormatMessage	*void ModelicaFormatMessage(const char* string, ...)* Output the message under the same format control as the C-function printf.
ModelicaError	*void ModelicaError(const char* string)* Output the error message string (no format control). This function never returns to the calling function, but handles the error similarly to an assert in the Modelica code.
ModelicaFormatError	*void ModelicaFormatError(const char* string, ...)* Output the error message under the same format control as the C-function printf. This function never returns to the calling function, but handles the error similarly to an assert in the Modelica code.
ModelicaAllocateString	*char* ModelicaAllocateString(size_t len)* Allocate memory for a Modelica string which is used as return argument of an external Modelica function. Note, that the storage for string arrays (= pointer to string array) is still provided by the calling program, as for any other array. If an error occurs, this function does not return, but calls "ModelicaError".
ModelicaAllocateStringWithErrorReturn	*char* ModelicaAllocateStringWithErrorReturn(size_t len)* Same as ModelicaAllocateString, except that in case of error, the function returns 0. This allows the external function to close files and free other open resources in case of error. After cleaning up resources use ModelicaError or ModelicaFormatError to signal the error.

6.7 External objects

External functions may have internal memory reported between function calls. Within Modelica this memory is defined as instance of the predefined class **ExternalObject** according to the following rules:

- There is a predefined **partial** class "ExternalObject″
  [since the class is partial, it is not possible to define an instance of this class].
- An external object class shall be directly extended from "ExternalObject", shall have exactly two function definitions, called "constructor" and "destructor", and shall not contain other elements.
- The constructor function is called exactly once before the first use of the object. For each completely constructed object, the destructor is called exactly once, after the last use of the object, even if an error occurs. The constructor shall have exactly one output argument in which the constructed ExternalObject is returned. The destructor shall have no output arguments and the only input argument of the destructor shall be the ExternalObject. It is not legal to call explicitly the constructor and destructor functions.
- Classes derived from ExternalObject can neither be used in an extends clause nor in a short class definition.
- External functions may be defined which operate on the internal memory of an ExternalObject. An ExternalObject used as input argument or return value of an external C-function is mapped to the C-type "`void*`".

[Example:

A user-defined table may be defined in the following way as an ExternalObject
(the table is read in a user-defined format from file and has memory for the last used table interval):

```
class MyTable
   extends ExternalObject;
   function constructor
     input  String  fileName = "";
     input  String  tableName = "";
     output MyTable table;
     external "C" table = initMyTable(fileName, tableName);
   end constructor;

   function destructor "Release storage of table"
     input  MyTable table;
     external "C" closeMyTable(table);
   end destructor;
end MyTable;
```

and used in the following way:

```
model test "Define a new table and interpolate in it"
   MyTable table=MyTable(fileName ="testTables.txt",
                         tableName="table1");  // call initMyTable
   Real y;
equation
   y = interpolateMyTable(table, time);
end test;
```

This requires to provide the following Modelica function:

```
function interpolateMyTable "Interpolate in table"
  input  MyTable table;
  input  Real  u;
  output Real  y;
  external "C" y = interpolateMyTable(table, u);
end interpolateTable;
```

The external C-functions may be defined in the following way:

```
typedef struct {  /* User-defined datastructure of the table */
  double* array;      /* nrow*ncolumn vector       */
  int     nrow;       /* number of rows            */
  int     ncol;       /* number of columns         */
  int     type;       /* interpolation type        */
  int     lastIndex;  /* last row index for search */
} MyTable;

void* initMyTable(char* fileName, char* tableName) {
  MyTable* table = malloc(sizeof(MyTable));
  if ( table == NULL ) ModelicaError("Not enough memory");
  // read table from file and store all data in *table
  return (void*) table;
};

void closeMyTable(void* object) { /* Release table storage */
  MyTable* table = (MyTable*) object;
  if ( object == NULL ) return;
  free(table->array);
  free(table);
}

double interpolateMyTable(void* object, double u) {
  MyTable* table = (MyTable*) object;
  double y;
  // Interpolate using "table" data (compute y)
  return y;
};
```

]

7 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. The only requirement is that any tool must be able to save files with all annotations intact, including those that are not used. 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.

7.1 Annotations for documentation

```
documentation_annotation:
    annotation"(" Documentation "(" "info" "=" STRING
                            ["," "revisions" "=" STRING ] ")" ")"
```

The “Documentation” annotation can contain the “info” annotation giving a textual description, the “revisions” annotation giving a list of revisions and other annotations defined by a tool *[The “revisions” documentation may be omitted in printed documentation]*. How the tool interprets the information in “Documentation” is unspecified. Within a string of the “Documentation” annotation, the tags <HTML> and </HTML> or <html> and </html> define optionally begin and end of content that is HTML encoded. Links to Modelica classes may be defined with the HTML link command using scheme “Modelica”, e.g.,

```
<a href="Modelica://MultiBody.Tutorial">MultiBody.Tutorial</a>
```

Together with scheme “Modelica” the (URI) fragment specifiers #diagram, #info, #text, #icon may be used to reference different layers. Example:

```
<a href="Modelica://MultiBody.Joints.Revolute#info">Revolute</a>
```

7.2 Annotations for graphical objects

A graphical representation of a class consists of two abstraction layers, icon layer and diagram layer showing graphical objects, component icons, connectors and connection lines. The icon representation typically vizualizes the component by hiding hierarchical details. The hierarchical decomposition is typically described in the diagram layer showing icons of subcomponents.

Graphical annotations described in this chapter ties into the Modelica grammar as follows.

```
graphical_annotations :
    annotation "(" [ layer_annotations ] ")"

layer_annotations:
    ( icon_layer | diagram_layer ) ["," layer_annotations ]
```

Layer descriptions (start of syntactic description):

```
icon_layer :
        "Icon" "(" [ coordsys_specification "," ] graphics ")"

diagram_layer :
        "Diagram" "(" [ coordsys_specification "," ] graphics ")"
```

[Example:

```
annotation (
  Icon(coordinateSystem(extent={{-10,-10}, {10,10}}),
       graphics={Rectangle(extent={{-10,-10}, {10,10}}),
                 Text({{-10,-10}, {10,10}}, textString="Icon")}));
```

]

The graphics is specified as an ordered sequence of graphical primitives, which are described below. *[Note that the ordered sequence is syntactically a valid Modelica annotation, although there is no mechanism for defining an array of heterogeneous objects in Modelica.]*

7.2.1 Common definitions

The following common definitions are used to define graphical annotations in the later sections.

```
type DrawingUnit = Real(final unit="mm");
type Point       = DrawingUnit[2]   "{x, y}";
type Extent      = Point[2]
                   "Defines a rectangular area {{x1, y1}, {x2, y2}}";
```

The interpretation of "unit" is with respect to printer output in natural size (not zoomed). *[On the screen, 1 mm in "natural size" is typically mapped to 4 pixels.]*

All graphical entities have a visible attribute which indicates if the entity should be shown.

```
partial record GraphicItem
   Boolean visible = true;
end GraphicItem;
```

7.2.1.1 Coordinate systems

Each of the layers have their own coordinate system. A coordinate system is defined by the coordinates of two points, at the lower left corner and at the upper right corner.

```
record CoordinateSystem    "Attribute to layer"
    Extent extent;
end CoordinateSystem;
```

[A coordinate system for an icon could for example be defined as:

```
CoordinateSystem(extent = {{-10, -10}, {10, 10}});
```

i.e. a first quadrant coordinate system with width 20 units and height 20 units. The exact interpretation of the units is to a certain extent tool dependent.]

The coordinate systems for icon and diagram are by default defined as follows; the array of GraphicsItem represents an ordered list of graphical primitives.

```
record Icon                 "Representation of Icon layer"
   CoordinateSystem  coordinateSystem(extent =
      {{-10, -10}, {10, 10}});
   GraphicItem[:] graphics;
end Icon;

record Diagram              "Representation of Diagram layer"
   CoordinateSystem  coordinateSystem(extent =
      {{-100, -100}, {100, 100}});
   GraphicItem[:] graphics;
end Diagram;
```

[A coordinate system for a connector icon could for example be defined as:

```
CoordinateSystem(extent = {{-1, -1}, {1, 1}});
```

]

7.2.1.2 Graphical properties

Properties of graphical objects and connection lines are described using the following attribute types.

```
type Color = Integer[3](min=0, max=255)   "RGB representation";

constant Color Black = zeros(3);

type LinePattern = enumeration(None, Solid, Dash, Dot, DashDot,
   DashDotDot);

type FillPattern = enumeration(None, Solid, Horizontal, Vertical,
   Cross, Forward, Backward, CrossDiag, HorizontalCylinder,
   VerticalCylinder, Sphere);

type BorderPattern = enumeration(None, Raised, Sunken, Engraved);
```

The FillPattern attributes Horizontal, Vertical, Cross, Forward, Backward and CrossDiag specify fill patterns drawn with the border color over the fillColor.

The attributes HorizontalCylinder, VerticalCylinder and Sphere specify gradients that represent a horizontal

cylinder, a vertical cylinder and a sphere, respectively.

The border pattern attributes Raised and Sunken represent panels which are rendered in a system-dependent way. The boder pattern Engraved represents a system-dependent outline.

```
type Arrow = enumeration(None, Open, Filled, Half);

type TextStyle = enumeration(Bold, Italic, UnderLine);
```

Filled shapes have the following attributes for the border and interior.

```
record FilledShape   "Style attributes for filled shapes"
   Color lineColor = Black        "Color of border line";
   Color fillColor = Black        "Interior fill color";

   LinePattern pattern = LinePattern.Solid   "Border line pattern";
   FillPattern fillPattern = FillPattern.None
                             "Interior fill pattern";

DrawingUnit lineThickness = 0.25 "Border line thickness"
end Style;
```

When color gradient is specified, the color fades from the specified fill color to white and black using the hue and saturation of the specified color.

7.2.2 Component instance and extends clause

A component instance and an extends clause can be placed within a diagram or icon layer. It has an annotation with a Placement modifier to describe the placement. Placements are defined in term of coordinate systems transformations:

```
record Transformation
   DrawingUnit x=0, y=0;
   Real scale=1, aspectRatio=1;
   Boolean flipHorizontal=false, flipVertical=false;
   Real rotation(quantity="angle", unit="deg")=0;
end Transformation;
```

The coordinates {x,y} define the position of the origin of the component's icon coordinate system. The scale attribute defines a uniform scale factor of the icon's coordinate system when draw in the coordinate system of the enclosing class. The aspectRatio attribute defines non-uniform scaling, where sy=aspectRatio*sx. The flipHorizontal and flipVertical attributes define horizontal and vertical flipping around the coordinate system axes. The graphical operations are applied in the order: scaling, flipping and rotation.

```
record Placement          "Attribute for component and extends"
   extends GraphicItem;
   Transformation transformation;
   Transformation iconTransformation  "Placement in icon layer";
end Placement;
```

A connector can be shown in both an icon layer and a diagram layer of a class. Since the coordinate systems typically are different, placement information needs to be given using two different coordinate systems. More flexibility than just using scaling and translation is needed since the abstraction views might need different visual placement of the connectors. *Placement* gives the placement in the diagram layer and *iconPlacement* gives the placement in an icon layer. When a connector is shown in a diagram layer, it's diagram layer is shown to facilitate opening up a hierarchical connector to allow connections to it's internal subconnectors.

For connectors, the icon layer is used to represent a connector when it is shown in the icon layer of the enclosing model. The diagram layer of the connector is used to represent it when shown in the diagram layer of the enclosing model. Protected connectors are only shown in the diagram layer. Public connectors are shown in both the diagram layer and the icon layer. Non-connector components are only shown in the diagram layer.

7.2.3 Connections

A connection is specified with an annotation containing a `Line` primitive, as specified below. *[Example:*

```
connect(a.x, b.x)
  annotation(Line(points={{-25,30}, {10,30}, {10, -20}, {40,-20}}));
```

];

7.2.4 Graphical primitives

This section describes the graphical primitives that can be used to define the graphical objects in an annotation.

7.2.4.1 Line

A line is specified as follows:

```
record Line
   extends GraphicItem;

   Point points[:];

   Color color = Black;
   LinePattern pattern = LinePattern.Solid;
   DrawingUnit thickness = 0.25;

   Arrow arrow[2] = {Arrow.None, Arrow.None};   "{start arrow, end arrow}"
   DrawingUnit arrowSize=3;
   Boolean smooth = false                    "Spline";
end Line;
```

Note that the `Line` primitive is also used to specify the graphical representation of a connection.

7.2.4.2 Polygon

A polygon is specified as follows:

```
record Polygon
   extends GraphicItem;
   extends FilledShape;
```

```
   Point points[:];
   Boolean smooth = false          "Spline outline";
end Polygon;
```

The polygon is automatically closed, if the first and the last points are not identical.

7.2.4.3 Rectangle

A rectangle is specified as follows:

```
record Rectangle
   extends GraphicItem;
   extends FilledShape;
   BorderPattern borderPattern = BorderPattern.None;
   Extent extent;
   DrawingUnit radius = 0                  "Corner radius";
end Rectangle;
```

The extent specifies the bounding box of the rectangle. If radius is specified, the rectangle is drawn with rounded corners of the given radius.

7.2.4.4 Ellipse

An ellipse is specified as follows:

```
record Ellipse
   extends GraphicItem;
   extends FilledShape;
   Extent extent;
end Ellipse;
```

The extent specifies the bounding box of the ellipse.

7.2.4.5 Text

A text string is specified as follows:

```
record Text
  extends GraphicItem;
  extends FilledShape;
  Extent extent;
  String textString;
  DrawingUnit fontSize;
  String fontName;
  TextStyle textStyle[:];
end Text;
```

The style attribute fontSize specifies the font size. If the fontSize attribute is 0 the text is scaled to fit its extent. Otherwise, the size specifies the absolute size. *[Note: the unit "point" is 1/72 inch, approximately 0.35 mm.]*

The style attribute textStyle specifies variations of the font.

7.2.4.6 Bitmap

A bitmap image is specified as follows:

```
record BitMap
   extends GraphicItem;
   Extent extent;
   String fileName                "Name of bitmap file";
   String imageSource             "Pixmap representation of bitmap";
end BitMap;
```

The bitmap primitive renders a graphical bitmap image. The data of the image can either be stored on an external file or in the annotation itself. The image is scaled to fit the extent.

When the attribute `fileName` is specified, the string refers to an external file containing image data. The mapping from the string to the file is unspecified. The supported file formats include PNG, BMP and JPEG, other supported file formats are unspecified.

When the attribute `imageSource` is specified, the string contains the image data. The image is representation the Pixmap format. *[Note: the Pixmap format is well defined and can be used both as a file format and embedded as a string. There are public-domain libraries for reading and writing Pixmap files.]*

The image is uniformly scaled [to preserve aspect ratio] so it exactly fits within the extent [touching the extent along one axis]. The center of the image is positioned at the center of the extent.

7.3 Annotations for the graphical user interface

A class may have the following annotations to define properties of the graphical user interface:

- **annotation**(defaultComponentName = "*name*")
  When creating a component of the given class, the recommended component name is *name*.
- **annotation**(defaultComponentPrefixes = "*prefixes*")
  When creating a component, it is recommended to generate a declaration of the form
  prefixes class-name component-name
  The following prefixes may be included in the string *prefixes*: inner, outer, replaceable, constant, parameter, discrete. *[In combination with defaultComponentName it can be used to make it easy for users to create inner components matching the outer declarations; see also example below]*
- **annotation**(missingInnerMessage = "*message*")
  When an "outer" component of the class does not have a corresponding "inner" component, the string *message* may be used as a diagnostic message.

[Example:

```
model World
  annotation(defaultComponentName     = "world",
             defaultComponentPrefixes = "inner replaceable",
             missingInnerMessage = "The World object is missing");
  ...
end World;
```

When an instance of model World is dragged in to the diagram layer, the following declaration is generated

```
   inner replaceable World world;
]
```

A declaration may have the following annotations

- **annotation**(unassignedMessage = "*message*");
  When the variable to which this annotation is attached in the declaration cannot be computed due to the structure of the equations, the string *message* can be used as a diagnostic message. *[When using BLT partitioning, this means if a variable "a" or one of its aliases "b = a", "b = -a", cannot be assigned, the message is displayed. This annotation is used to provide library specific error messages.]*
- **annotation**(Dialog(enable = *parameter-expression*,
  tab = "*tab*", group = "*group*"));
  Defines the placement of the component or class parameter in a parameter dialog with optional tab and group specification. If enable is **false**, the input field may be disabled *[and no input can be given]*. "Dialog" is defined as:

```
record Dialog
  parameter String  tab    = "General";
  parameter String  group  = "Parameters";
  parameter Boolean enable = true;
end Dialog;
```

  A parameter dialog is a sequence of tabs with a sequence of groups inside them.

```
[Examples:
      connector Frame "Frame of a mechanical system"
          ...
        flow Modelica.SIunits.Force f[3]
           annotation(unassignedMessage = "
      All Forces cannot be uniquely calculated. The reason could be that
      the mechanism contains a planar loop or that joints constrain the
      same motion. For planar loops, use in one revolute joint per loop the
      option PlanarCutJoint=true in the Advanced menu.
      ");
      end Frame;


      model BodyShape
        ...
        parameter Boolean animation = true;
        parameter SI.Length length "Length of shape"
           annotation(Dialog(enable = animation, tab = "Animation",
                             group = "Shape definition"));
        ...
      end BodyShape;
]
```

7.4 Annotations for version handling

A top-level package or model can specify the version of top-level classes it uses, its own version number, and if possible how to convert from previous versions. This can be used by a tool to guarantee that consistent versions

are used, and if possible to upgrade usage from an earlier version to a current one.

7.4.1 Version numbering

Version numbers are of the forms:

- Main release versions: """ UNSIGNED_INTEGER { "." UNSIGNED_INTEGER } """
  [Example: "2.1"]
- Pre-release versions: """ UNSIGNED_INTEGER { "." UNSIGNED_INTEGER } " " {S-CHAR} """
  [Example: "2.1 Beta 1"]
- Un-ordered versions: """ NON-DIGIT {S-CHAR} """
  [Example: "Test 1"]

The main release versions are ordered using the hierarchical numerical names, and follow the corresponding pre-release versions. The pre-release versions of the same main release version are internally ordered alphabetically.

7.4.2 Version handling

In a top-level class, the version number and the dependency to earlier versions of this class are defined using one or more of the following annotations:

- version = CURRENT-VERSION-NUMBER
  Defines the version number of the model or package. All classes within this top-level class have this version number.
- conversion (noneFromVersion = VERSION-NUMBER) Defines that user models using the VERSION-NUMBER can be upgraded to the CURRENT-VERSION-NUMBER of the current class without any changes.
- conversion (from (version = VERSION-NUMBER, script = "…"))
  Defines that user models using the VERSION-NUMBER can be upgraded to the CURRENT-VERSION-NUMBER of the current class by applying the given script. The semantics of the conversion script is not defined.
- uses(IDENT (version = VERSION-NUMBER))
  Defines that classes within this top-level class uses version VERSION-NUMBER of classes within the top-level class IDENT.

The annotations uses and conversion may contain several different sub-entries.

[Example:

```
package Modelica
  annotation(version=”2.1”,
             conversion(noneFromVersion=”2.1 Beta 1”,
                        from(version=”1.5”,
                            script=”convertFromModelica1_5.mos”)));
  ...
end Modelica;

model A
  annotation(version="1.0",
     uses(Modelica(version="1.5")));
```

```
  ...
end A;

model B
  annotation(uses(Modelica(version="2.1 Beta 1")));
  ...
end B;
```

In this example the model A uses an older version of the Modelica library and can be upgraded using the given script, and model B uses an older version of the Modelica library but no changes are required when upgrading.]

7.4.3 Mapping of versions to file system

A top-level class, IDENT, with version VERSION-NUMBER can be stored in one of the following ways in a directory given in the MODELICAPATH:

- The file IDENT ".mo" *[Example: Modelica.mo]*
- The file IDENT " " VERSION-NUMBER ".mo" *[Example: Modelica 2.1.mo]*
- The directory IDENT *[Example: Modelica]*
- The directory IDENT " " VERSION-NUMBER *[Example: Modelica 2.1]*

This allows a tool to access multiple versions of the same package.

8 Modelica standard library

The pre-defined, free "**package** Modelica" is shipped together with a Modelica translator. It is an extensive standard library of pre-defined components in several domains. Futhermore, it contains a standard set of **type** and **interface** definitions in order to influence the trivial decisions of model design process. If, as far as possible, standard quantity types and connectors are relied on in modeling work, model compatibility and thereby reuse is enhanced. Achieving model compatibility, without having to resort to explicit coordination of modeling activities, is essential to the formation of globally accessible libraries. Naturally, a modeller is not required to use the standard library and may add any number of local base definitions.

The library will be amended and revised as part of the ordinary language revision process. It is expected that informal standard base classes will develop in various domains and that these gradually will be incorporated into the Modelica standard library.

The type definitions in the library are based on ISO 31-1992. Several ISO quantities have long names that tend to become awkward in practical modeling work. For this reason, shorter alias-names are also provided if necessary. Using, e.g., "ElectricPotential" repeatedly in a model becomes cumbersome and therefore "Voltage" is supplied as an alternative.

The standard library is not limited to pure SI units. Whenever common engineering practice uses a different set of (possibly inconsistent) units, corresponding quantities will be allowed in the standard library, for example English units. It is also frequently common to write models with respect to scaled SI units in order to improve the condition of the model equations or to keep the actual values around one for easier reading and writing of numbers.

The connectors and partial models have predefined graphical attributes in order that the basic visual appearance is the same in all Modelica based systems.

The complete Modelica package can be downloaded from http://www.Modelica.org/library/library.html. Below, the introductory documentation of this library is given. Note, that the Modelica package is still under development.

```
package Modelica
  annotation(Documentation(info="
    /* The Modelica package is a standardized, pre-defined and free
       package, that is shipped together with a Modelica translator. The
       package provides constants, types, connectors, partial models and
       model components in various disciplines.

       In the Modelica package the following conventions are used:

       - Class and instance names are written in upper and lower case
         letters, e.g., "ElectricCurrent". An underscore is only used
         at the end of a name to characterize a lower or upper index,
         e.g., body_low_up.

       - Type names start always with an upper case letter.
```

```
      Instance names start always with a lower case letter with only
      a few exceptions, such as "T" for a temperature instance.

    - A package XXX has its interface definitions in subpackage
      XXX.Interface, e.g., Electrical.Interface.


    - Preferred instance names for connectors:
      p,n: positive and negative side of a partial model.
      a,b: side "a" and side "b" of a partial model
           (= connectors are completely equivalent).
 */
 "));
end Modelica;
```

9 Revision history

This section describes the history of the Modelica Language Design, and its contributors. The current version of this document is available from http://www.modelica.org/.

9.1 Modelica 2.1

Modelica 2.1 was released January 30, 2004. The Modelica 2.1 specification was edited by Hans Olsson and Martin Otter.

9.1.1 Contributors to the Modelica Language, version 2.1

Mikael Adlers, MathCore, Linköping, Sweden
Peter Aronsson, Linköping University, Sweden
Bernhard Bachmann , University of Applied Sciences, Bielefeld, Germany
Peter Bunus, Linköping University, Sweden
Jonas Eborn, United Technologies Research Center, Hartford, U.S.A.
Hilding Elmqvist, Dynasim, Lund, Sweden
Rüdiger Franke, ABB Corporate Research, Ladenburg, Germany
Peter Fritzson, Linköping University, Sweden
Anton Haumer, Technical Consulting & Electrical Engineering, St.Andrae-Woerdern, Austria
Olof Johansson, Linköping University, Sweden
Karin Lunde, R.O.S.E. Informatik GmbH, Heidenheim, Germany
Sven Erik Mattsson, Dynasim, Lund, Sweden
Hans Olsson, Dynasim, Lund, Sweden
Martin Otter, German Aerospace Center, Oberpfaffenhofen, Germany
Levon Saldamli, Linköping University, Sweden
Christian Schweiger, German Aerospace Center, Oberpfaffenhofen, Germany
Michael Tiller, Ford Motor Company, Dearborn, MI, U.S.A.
Hubertus Tummescheit, United Technologies Research Center, Hartford, U.S.A.
Hans-Jürg Wiesmann, ABB Switzerland Ltd.,Corporate Research, Baden, Switzerland

9.1.2 Main changes in Modelica 2.1

The main changes in Modelica 2.1 are:

- Arrays and array indices of Enumerations (needed, e.g., in the Electrical.Digital library currently under development).
- Connections into hierarchical connectors (needed, e.g., for convenient implementation of buses).
- Optional output arguments of Modelica functions. The presence of actual input and/or output arguments can be inquired with the new built-in function *isPresent*(..). The previous built-in function and attribute *enable*

was removed.

- Making the default constraining type more useful by inheriting the base constraining type automatically to modifications.
- Enhanced redeclaration as needed, e.g., in the Modelica.Media library under development (e.g. "*redeclare model* name" or "*model extends* name (<modifications>)").
- Handling of overdetermined connectors (needed, e.g., for multi-body systems and electrical power systems) including the new built-in package *Connections* with operators *Connections.branch*, *Connections.root*, *Connections.potentialRoot*, *Connections.isRoot*.
- Statement *break* in the while loop of an algorithm section.
- Statement *return* in a Modelica function.
- Built-in function *String*(..) to provide a string representation of Boolean, Integer, Real and Enumeration types.
- Built-in function *Integer*(..) to provide the Integer representation of an Enumeration type.
- Built-in function *semiLinear*(..) to define a characteristics with two slopes and a set of rules for symbolic transformations, especially when the function becomes underdetermined (this function is used in the Modelica Fluid library under development to define reversing flow in a mathematically clean way).
- More general identifiers by having any character in single quotes, e.g. ′+′ or ′123.456#1′ are valid identifiers. ′x′ and x are different identifiers. This is useful for a direct mapping of product identifiers to model names and for having the usual symbols for digital electrical signals as enumerations (such as ′+′, ′-′, ′0′, ′1′).
- New annotations:
  - For version handling of libraries and models (*version, uses, conversion*),
  - for revision logging (*revisions*),
  - for using a Modelica name as link in a HTML documentation text,
  - for convenient "inner" declaration in a GUI (*defaultComponentName, defaultComponentPrefixes*),
  - for parameter menu structuring (*Dialog, enable, tab, group*), and
  - for library specific error messages (*missingInnerMessage, unassignedMessage*).
- Fixing some minor errors in the grammar and semantic specification.

The language changes are backward compatible, except for the introduction of the new keywords *break* and *return*, the new built-in package *Connections* and the removing of built-in function and attribute *enable*.

9.2 Modelica 2.0

Modelica 2.0 was released January, 30 2002, and the draft was released on December 18 in 2001. The Modelica 2.0 specification was edited by Hans Olsson. Modelica is a ***registered trademark*** owned by the Modelica Association since November 2001.

9.2.1 Contributors to the Modelica Language, version 2.0

Peter Aronsson, Linköping University, Sweden
Bernhard Bachmann , University of Applied Sciences, Bielefeld
Peter Beater, University of Paderborn, Germany
Dag Brück, Dynasim, Lund, Sweden

Peter Bunus, Linköping University, Sweden
Hilding Elmqvist, Dynasim, Lund, Sweden
Vadim Engelson, Linköping University, Sweden
Peter Fritzson, Linköping University, Sweden
Rüdiger Franke, ABB Corporate Research, Ladenburg
Pavel Grozman, Equa, Stockholm, Sweden
Johan Gunnarsson, MathCore, Linköping
Mats Jirstrand, MathCore, Linköping
Sven Erik Mattsson, Dynasim, Lund, Sweden
Hans Olsson, Dynasim, Lund, Sweden
Martin Otter, German Aerospace Center, Oberpfaffenhofen, Germany
Levon Saldamli, Linköping University, Sweden
Michael Tiller, Ford Motor Company, Dearborn, MI, U.S.A.
Hubertus Tummescheit, Lund Institute of Technology, Sweden
Hans-Jürg Wiesmann, ABB Switzerland Ltd.,Corporate Research, Baden, Switzerland

9.2.2 Main changes in Modelica 2.0

A detailed description of the enhancements introduced by Modelica 2.0 are given in the papers

- M. Otter, H. Olsson: ***New Features in Modelica 2.0***. 2nd International Modelica Conference, March 18-19, DLR Oberpfaffenhofen, Proceedings, pp. 7.1 - 7.12, 2002. This paper can be downloaded from http://www.Modelica.org/Conference2002/papers/p01_Otter.pdf
- Mattsson S. E., Elmqvist H., Otter M., and Olsson H.: ***Initialization of Hybrid Differential-Algebraic Equations in Modelica 2.0***. 2nd International Modelica Conference, March 18-19, DLR Oberpfaffenhofen, Proceedings, pp. 9 - 15, 2002. This paper can be downloaded from http://www.Modelica.org/Conference2002/papers/p02_Mattsson.pdf

The main changes in Modelica 2.0 are:

- Full specification of initialization in order to compute consistent initial values of all variables appearing in a model before performing an operation, such as simulation or linearization.
- Specified the graphical appearance of Modelica object diagrams, thereby ensuring portability of model topology information and improving the previous informal graphical description, e.g., with separate icon and diagram positions.
- Enumeration types to allow the definition of options and properties in an understandable, safe and efficient way.
- Support for (optional) explicit preference in state-selection in order that a modeler can incorporate application specific knowledge to guide the solution process, e.g., for real-time simulation.
- Iterators in array constructors and reduction operators, to support more powerful expressions, especially in declarations, in order to avoid inconvenient and less efficient local function definitions.
- Support for generic formulation of blocks applicable to both scalar and vector connectors, connection of (automatically) vectorized blocks, and simpler input/output connectors. This allows significant simplifications of the input/output block library of Modelica, e.g., since only scalar versions of all blocks have to be provided. Furthermore, new library components can be incorporated more easily.
- Record constructor to allow, e.g., the construction of data sheet libraries.

- Functions with mixed positional and named arguments. Optional results and default arguments make the same function fit for beginners and expert users.
- Additional utilities for external C-functions that are interfaced to Modelica models, especially supporting external functions returning strings and external functions with internal memory (e.g., to interface user-defined tables, property databases, sparse matrix handling, hardware interfaces).
- Added an index, and specification of some basic constructs that had previously not formally be defined, such as while-clauses, if-clauses.

The language changes are backward compatible, except for the introduction of the new keyword enumeration. The library change of the block library which will become available soon requires changes in user-models.

9.3 Modelica 1.4

Modelica 1.4 was released December 15, 2000. The Modelica Association was formed in Feb. 5, 2000 and is now responsible for the design of the Modelica language. The Modelica 1.4 specification was edited by Hans Olsson and Dag Brück.

9.3.1 Contributors to the Modelica Language, version 1.4

Bernhard Bachmann, Fachhochschule Bielefeld, Germany
Peter Bunus, MathCore, Linköping, Sweden
Dag Brück, Dynasim, Lund, Sweden
Hilding Elmqvist, Dynasim, Lund, Sweden
Vadim Engelson, Linköping University, Sweden
Jorge Ferreira, University of Aveiro, Portugal
Peter Fritzson, Linköping University, Linköping, Sweden
Pavel Grozman, Equa, Stockholm, Sweden
Johan Gunnarsson, MathCore, Linköping, Sweden
Mats Jirstrand, MathCore, Linköping, Sweden
Clemens Klein-Robbenhaar, Germany
Pontus Lidman, MathCore, Linköping, Sweden
Sven Erik Mattsson, Dynasim, Lund, Sweden
Hans Olsson, Dynasim, Lund, Sweden
Martin Otter, German Aerospace Center, Oberpfaffenhofen, Germany
Tommy Persson, Linköping University, Sweden
Levon Saldamli, Linköping University, Sweden
André Schneider, Fraunhofer Institute for Integrated Circuits, Dresden, Germany
Michael Tiller, Ford Motor Company, Dearborn, MI, U.S.A.
Hubertus Tummescheit, Lund Institute of Technology, Sweden
Hans-Jürg Wiesmann, ABB Corporate Research Ltd., Baden, Switzerland

9.3.2 Contributors to the Modelica Standard Library

Peter Beater, University of Paderborn, Germany
Christoph Clauß, Fraunhofer Institute for Integrated Circuits, Dresden, Germany
Martin Otter, German Aerospace Center, Oberpfaffenhofen, Germany
André Schneider, Fraunhofer Institute for Integrated Circuits, Dresden, Germany
Hubertus Tummescheit, Lund Institute of Technology, Sweden

9.3.3 Main Changes in Modelica 1.4

- Removed declare-before-use rule. This simplifies graphical user environments, because there exists no order of declarations when components are graphically composed together.
- Refined package concept by introducing encapsulated classes and import mechanism. Encapsulated classes can be seen as "self-contained units": When copying or moving an encapsulated class, at most the import statements in this class have to be changed.
- Refined when-clause: The nondiscrete keyword is removed, equations in when-clauses must have a unique variable name on left hand side variable and the exact mapping of when-clauses to equations is defined. As a result, when-clauses are now precisely defined without referring to a sorting algorithm and it is possible to handle algebraic loops between when-clauses with different conditions and between when-clauses and the continuous-time part of a model. The discrete keyword is now optional, simplifying the library development because only one type of connector is needed and not several types which do contain or do not contain the discrete prefix on variables. Additionally, when-clauses in algorithm sections may have elsewhen clauses which simplifies the definition of priorities between when-clauses.
- For replaceable declarations: allowed constraining clauses, and annotations listing suitable redeclarations. This allows a graphical user environment to automatically build menus with meaningful choices.
- Functions can specify their derivative. This allows, e.g., the application of the Pantelides algorithm to reduce the index of a DAE also for external functions.
- New built-in operator "**rem**" (remainder) and the built-in operators **div**, **mod**, **ceil**, **floor**, **integer**, previously only allowed to be used in when-clauses can now be used everywhere, because state events are automatically generated when the result value of one of these operator changes discontinuously.
- Quantity attribute also for base types Boolean, Integer, String (and not only for Real), in order to allow abstracted variables to refer to physical quantities (e.g. Boolean i(quantity="Current") is true if current is flowing and is false if no current is flowing).
- final keyword also allowed in declaration, to prevent modification. Example

```
model A
  Real x[:];
  final Integer n=size(x,1);
end A;
```

- Several minor enhancements, such as usage of dot-notation in modifications (e.g.: `"A x(B.C=1,B.D=2)"` is the same as `"A x(B(C=1,D=2));")`.
- Internally restructured specification.

Modelica 1.4 is backwards compatible with Modelica 1.3, with the exception of (1) some exotic cases where different results are achieved with the removed "declare-before-use-rule" and the previous declaration order, (2) when-clauses in equations sections, which use the general form "expr1 = expr2" (now only "v=expr" is allowed + some special cases for functions), (3) some exotic cases where a when-clause may be no longer evaluated at the initial time, because the initialization of the when-condition is now defined in a more meaningful way (before Modelica 1.4, every condition in a when-clause has a "previous" value of false), and (4) models containing the nondiscrete keyword which was removed.

9.4 Modelica 1.3 and older versions.

Modelica 1.3 was released December 15, 1999.

9.4.1 Contributors up to Modelica 1.3

The following list contributors and their affiliations at the time when Modelica 1.3 was released.

H. Elmqvist[1],

B. Bachmann[2], F. Boudaud[3], J. Broenink[4], D. Brück[1], T. Ernst[5], R. Franke[6], P. Fritzson[7], A. Jeandel[3], P. Grozman[12], K. Juslin[8], D. Kågedal[7], M. Klose[9], N. Loubere[3], S. E. Mattsson[1], P. J. Mosterman[11], H. Nilsson[7], H. Olsson[1], M. Otter[11], P. Sahlin[12], A. Schneider[13], M. Tiller[15], H. Tummescheit[10], H. Vangheluwe[16]

[1] Dynasim AB, Lund, Sweden
[2] ABB Corporate Research Center Heidelberg
[3] Gaz de France, Paris, France
[4] University of Twente, Enschede, Netherlands
[5] GMD FIRST, Berlin, Germany
[6] ABB Network Partner Ltd. Baden, Switzerland
[7] Linköping University, Sweden
[8] VTT, Espoo, Finland
[9] Technical University of Berlin, Germany
[10] Lund University, Sweden
[11] DLR Oberpfaffenhofen, Germany
[12] Bris Data AB, Stockholm, Sweden
[13] Fraunhofer Institute for Integrated Circuits, Dresden, Germany
[14] DLR, Cologne, Germany
[15] Ford Motor Company, Dearborn, MI, U.S.A.
[16] University of Gent, Belgium

9.4.2 Main changes in Modelica 1.3

Modelica 1.3 was released December 15, 1999.

- Defined connection semantics for inner/outer connectors.
- Defined semantics for protected element.
- Defined that least variable variability prefix wins.
- Improved semantic definition of array expressions.
- Defined scope of for-loop variables.

9.4.3 Main changes in Modelica 1.2

Modelica 1.2 was released June 15, 1999.

- Changed the external function interface to give greater flexibility.
- Introduced inner/outer for dynamic types.
- Redefined final keyword to only restrict further modification.

- Restricted redeclaration to replaceable elements.
- Defined semantics for if-clauses.
- Defined allowed code optimizations.
- Refined the semantics of event-handling.
- Introduced fixed and nominal attributes.
- Introduced terminate and analysisType.

9.4.4 Main Changes in Modelica 1.1

Modelica 1.1 was released in December 1998.

Major changes:

- Specification as a separate document from the rationale.
- Introduced prefixes discrete and nondiscrete.
- Introduced pre and when.
- Defined semantics for array expressions.
- Introduced built-in functions and operators (only connect was present in Modelica 1.0).

9.4.5 Modelica 1.0

Modelica 1, the first version of Modelica, was released in September 1997, and had the language specification as a short appendix to the rationale.

10 Index