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

Language Specification

Version 1.4,

December 15, 2000

by the

Modelica Association

Abstract

This document defines the Modelica language, version 1.4, 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 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 Modelica language is specified by means of a set of rules for translating a model described in Modelica 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 are functions of all input and of all result variables (number of equations = number of basic result variables).
- Algorithm sections where every section is treated as a set of equations which are functions of 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 }
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 "\"
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 ]
```

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. Inside a comment, the sequence `<HTML> .... </HTML>` indicates HTML code which may be used by tools to facilitate model documentation.

Keywords and built-in operators of the Modelica language are written in bold face. Keywords are reserved words and may not be used as identifiers.

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 )
   IDENT class_specifier

class_specifier :
   string_comment composition end IDENT
   | "=" name [ array_subscripts ] [ class_modification ] comment

composition  :
   element_list
   { public element_list |
```

```
     protected element_list |
     equation_clause |
     algorithm_clause
   }
   [ external [ language_specification ]
              [ external_function_call ] ";" [ annotation ";" ] ]

language_specification :
   STRING

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

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

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

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

2.2.3 Extends

```
extends_clause :
   extends name [ class_modification ]

constraining_clause :
   extends_clause
```

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 :
   [ final ] component_reference modification string_comment

element_redeclaration :
   redeclare
( ( 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 :
  equation { equation ";" | annotation  ";" }

algorithm_clause :
   algorithm { algorithm ";" | annotation  ";" }

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

algorithm :
  ( component_reference ( ":=" expression | function_call )
    | "(" expression_list ")" ":=" component_reference function_call
    | conditional_equation_a
    | for_clause_a
    | while_clause
    | when_clause_a
    | assert_clause )
  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 IDENT in expression loop
     { equation ";" }
  end for

for_clause_a :
  for IDENT in expression loop
     { algorithm ";" }
  end for

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

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

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

connect_clause :
  connect "(" connector_ref "," connector_ref ")"

connector_ref :
  IDENT [ array_subscripts ] [ "." IDENT [ array_subscripts ] ]

assert_clause :
  assert "(" expression "," STRING { "+" STRING } ")"
  terminate "(" STRING { "+" STRING } ")"
```

2.2.7 Expressions

```
expression :
   simple_expression
 | if 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 ]
 | "(" expression_list ")"
 | "[" expression_list { ";" expression_list } "]"
 | "{" expression_list "}"

name :
   IDENT [ "." name ]

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

function_call :
   "(" function_arguments ")"
```

```
function_arguments :
   expression_list
 | named_arguments

named_arguments: [named_argument { "," named_argument }]

named_argument: IDENT "=" 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 and C2 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 and C2. 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.

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.
- 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.
- 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 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

[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 C1 x4(a=4);    // x4.a has default value 4
  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(a=44), 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.a	*44*
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:

- *For functions with more than one input variable called with positional arguments, section 3.4.7.*
- *For functions with more than one output variable, section 3.4.7.*
- *Records that are used as arguments to external functions, section 6.2.3.*

]

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 are identical after instantiation.

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 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 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

3.1.4 External representation of classes

Classes may be represented in the hierarchical structure of the operating system *[the file system or a database]*. 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. 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.

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.

Components of function type may be instantiated. *[A modifier can be used to e.g. change parameters of the function. It is also possible to do such a modification with a class specialization.]* 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. 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.

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):

- Variables vc declared with the **parameter or constant** prefixes remain constant during transient analysis (vc=**const**.).
- **Discrete-time** variables vd have 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 their values only at event instants during transient analysis (see section 3.5).
- **Continuous-time** variables vn may have a non-vanishing time derivative (**der**(vn)≠0 possible) and may change their values 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.

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, or Boolean variables is discrete-time, and it is not possible to declare continuous-time Integer, String, or Boolean variables. *[A Modelica translator is able to guarantee this property due to restrictions imposed on discrete expressions, see section 3.4.8]*

The variability of expressions and restrictions on variability for definition equations is given in section 3.4.8.

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

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

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

```
  end Inertia;
```

*A **constant** variable is similar to a parameter with the difference that constants cannot be changed after they have been declared. 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 **discretetime** 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

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).

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

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 and String:

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[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];
```

]

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) 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;
```

]

Note: In the previous versions of Modelica (Modelica 1.0 and 1.1), the **final** keyword had three different meanings depending on the situation where it was used. To simplify the semantics, in Modelica 1.4, **final** is only used in modifications and declarations to prevent further modifications and redeclarations. As a consequence, components have to be explicitly defined as **replaceable**, if they shall be redeclared except for restricting prefix and/or array dimensions (previously, this was the default and **final** was used to prevent redeclarations).

3.2.7 Short class definition

A class definition of the form

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

is identical to the longer form

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

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.]

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(unit="V");
  Voltage v1, v2;
end C1;
class C2
  extends C1(Voltage(unit="kV"));
end C2;
```

Instantiation of class C2 yields a local class Voltage with unit-modifier "kV". The variables v1 and v2 instantiate this local class and thus have unit "kV".]

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=3);      // 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=1, A(b=2));
end C2;
```

yields an instance with `bcomp.b=1`*, 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 replaces the declaration of a local class or component in the modified element with another declaration.

[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;
```

]

3.2.10.1 Constraining type

In an replaceable declaration the optional constraining_clause define a constraining type. *[It is recommended to not have modifiers in the constraining_clause.]* If the constraining_clause is not present the type of the declaration is also used as a constraining type.

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

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**

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, enumerated 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 with an optional order-attribute indicating the order of the derivative (default 1). This can influence simulation time and accuracy and can be applied to both functions written in Modelica and to external functions.

[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 is a nth derivative (n>=1) the derivative annotation indicates the (n+1)th derivative, and order=n+1.

The input arguments amended by 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 predefined type (Real,Boolean,Integer and String) 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.

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.
type	May only be extension to the predefined types, 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, at most one algorithm section. Calling a function requires either an algorithm section 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**.

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

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 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 is in scope inside the loop-construct and shall not be assigned to.

[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
```

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;
```

]

3.3.4 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.

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 Assert

The expression of an assert clause shall evaluate to true. *[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).]*

3.3.6 Terminate

The terminate function successfully terminates the analysis which was carried out. 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;
```

]

3.3.7 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 an element of the same class as the **connect** statement or an element of one of its components. 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 outer 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.]

3.3.7.1 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).

3.3.7.2 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.

3.4 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 3.4.7 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
```

]

3.4.2 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.
analysisType()	Returns the most appropriate analysis type for the context in which the model is used. The analysis type is returned as a string. The following return values are predefined: "dynamic": Solve initial value problem "static": Solve "static" problem where all derivatives are constant and time is fixed (e.g. trimming, equilibrium analysis) "linear": Transform continuous part of model in a linear system.
initial()	Returns true at the beginning of analysis (where time is equal to time.start).
terminal()	Returns true at the end of a successful analysis.
noEvent(expr)	Real elementary relations within expr are taken literally, i.e., no state or time event is triggered.
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 Boolean, Integer or Real, (b) y is a discrete-time expression (c) the operator is **not** applied in a **function** class. At the initial time pre(y) = y.start, i.e., the left limit of y is identical to the start value. For parameter and constant variables pre(p)=p.
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.
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.]*.
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.
cardinality(c)	Returns the number of (internal and external) occurrences of connector instance c in a connect statement as an Integer number.

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 or Boolean. These systems of equations can be solved by a global fix point iteration scheme, similarly to the event iteration, by fixing the Boolean and Integer 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.]

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.*

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

Bouncing ball:

```
    der(h) = v;
    der(v) = -g;

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

Self-initializing block:

```
   block PT1 " first order filter"
     parameter Real T "time constant";
     parameter Real k "gain";
     input  Real u;
     output Real y;
   protected
     Real x;
   equation
     der(x) = (u – x) / T;
         y  = k*x;

   algorithm
     when initial() then
       reinit(x, u);   // initialize, such that der(x) = 0.
     end when
   end PT1;

   model Test
     PT1 b1, b2, b3;
     input u;
   equation
     b1.u = u;
     connect(b1.y, b2.u);
     connect(b2.y, b3.u);
   end Test;
```

Given the input signal u, all 3 blocks b1, b2, b3 are initialized at their stationary value.]

*[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|*

*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.

*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;
      else if cardinality(p) == 0 then
         p.i = 0;
      else if cardinality(n) == 0 then
         n.i = 0;
      end if

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

]

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 expression s which has to be a subtype of Real, Integer, Boolean or String (n_i >= 0). The returned array has the same type as 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.
min(A)	Returns the smallest element of array expression A.
max(A)	Returns the largest element of array expression A.
sum(A)	Returns the sum of all the elements of array expression A.
product(A)	Returns the product of all the elements of array expression A.
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);
```

]

3.4.4 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 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.

3.4.4.3 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];
```

]

3.4.4.4 Vector Construction

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

The colon operator of *simple-expression* can be used instead of or in combination with this general constructor to construct Real and Integer 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 or Integer 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.

[Examples:

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

]

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.

[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}] .
- 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.

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

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.

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];
A*x = b;
x*A = b;                    // same as transpose([x])*A*b
[v]*transpose([v])          // outer product
v*M*v                       // scalar
tranpose([v])*M*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 of records and *m* is a component of that record, the expression *a.m* is interpreted as *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* arguments. The result is Boolean and is *true* or *false* if the relation is fulfilled or not, respectively.

In relations of the form v1 == v2 or v1 <> v2, v1 or v2 shall not be 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 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.]

3.4.6.11 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 Functions

There are two forms of function application, see section 2.2.7. In the first form,

```
f(3.5, 5.76)
```

the arguments are associated with the *[formal]* parameters according to their position in the argument list. Thus argument *i* is passed to parameter *i*, where the order of the parameters is given by the order of the component declarations in the function definition. The first input component is parameter number 1, the second input component is parameter number 2, and so on. When a function is called in this way, the number of arguments and parameters must be the same.

In the second form of function application,

```
g(x=3.5, y=5.76)
```

the parameters are explicitly associated with the arguments by means of equations in the argument list. Parameters that have default values need not be specified in the argument list.

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 f is defined as follows:

```
function f
   input Real x;
   input Real y;
   input Real z := 10.0;
   output Real r;
   ...
end f;
```

Then the following two applications are equivalent:

```
f(1.0, 2.0, 10.0)
f(y = 2.0, x = 1.0)
```

]

A function may have more than one output component, corresponding to multiple return values. When a function has a single return value, a function application is an expression whose value and type are given by the value and type of the output component.

The only way to call a function having more than one output component is to make the function call the RHS of an equation or assignment. In these cases, the LHS of the equation or assignment must be a list of component references within parentheses. 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 number of component references in the list must agree with the number of output components.

The type of each output parameter must agree with the type of the corresponding component references in the list on the LHS.

[Example. Suppose a function f is defined as follows:

```
function f
   input Real x;
   input Real y;
   output Real r1;
   output Real r2;
   output Real r3;
   ...
end f;
```

Then the following equation and assignment show the two possible ways of calling f:

```
   (x, y, z) = f(1.0, 2.0);
   (x, y, z) := f(1.0, 2.0);
```

]

The only permissible use of an expression in the form of a list of expressions in parentheses, is when it is used as the LHS of an equation or assignment where the RHS is an application of a function with more than one output component. In this case, the expressions in the list shall be component references.

[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 Variability of Expressions

Constant expressions are:

- Real, Integer, Boolean and String literals.
- Real, Integer, Boolean and String 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.
- Real, Integer, Boolean and String 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 and String 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.

If the value of a constant or parameter expression is either directly or indirectly used as structural expression (i.e. to compute the size of a component or for if-statements with unequal sizes of the branches) it is a quality-of-implementation issue whether any calls of non-builtin functions are allowed as subexpressions. *[The intention is to erase this restriction for Modelica 2.0.]*

Components declared as **constant** shall have an associated declaration equation with a constant expression. The value of a constant cannot be changed after its declaration.

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 or String equations outside of a when-clause, because then one of the two expressions is not discrete-time]*

[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 noEvent feature is propagated to all subrelations in the scope of the noEvent function.

[Example:

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

The if-expression is taken literally without inducing state events.

*The **noEvent** 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. Furthermore, the **noEvent** function is used to guard agains "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 a discrete-time variable


algorithm
     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:

```
algorithm
    when condition1 then
       close := true;
    end when;
algorithm
    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:

```
algorithm
    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. For example, "deadlock" between different when-clauses is present if there are algebraic loops between the equations of the when-clauses.]

3.6 Variable attributes of predefined types

The attributes of the predefined variable types are described below with Modelica syntax although they are predefined; redeclaration of any of these types is an error. It is furthermore not possible to combine extends from the predefined types with other components. The definitions use RealType, IntegerType, BooleanType and StringType 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
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;
```

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:
Trigger an event at start time + 10 s:

```
   parameter Real T0 = time.start + 10;
algorithm
   when time >= T0 then
      ...
   end when;
```

]

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 3.3.7.1).
- 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**. 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 ";" ]
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 *`	Not allowed.

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 to facilitate calling of C functions. Currently, returning strings from external C-functions is not supported.

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`

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

6.2.2 Arrays

Unless an explicit function call is present in the external declaration, an arrays 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, …, 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
T[*dim*$_1$, …, *dim*$_n$]	*T′* *	*T′*
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);
```

]

7 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
  package 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 Info;
end Modelica;
```

8 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/.

8.1 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.

8.1.1 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, Detroit, U.S.A.
Hubertus Tummescheit, Lund Institute of Technology, Sweden
Hans-Jürg Wiesmann, ABB Corporate Research Ltd., Baden, Switzerland

8.1.2 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

8.1.3 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.

8.2 Modelica 1.3 and older versions.

Modelica 1.3 was released December 15, 1999.

8.2.1 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, Detroit, U.S.A.
[16] University of Gent, Belgium

8.2.2 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.

8.2.3 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.

8.2.4 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).

8.2.5 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.