Modelica®——统一的面向对象物理系统建模语言

语言规范（V2.2）

Modelica 协会

2005 年 2 月 2 日

注：本文是在周凡利博士翻译的 V2.1 和朱恒伟老师翻译的 V2.2 基础上校对整理的。

Modelica® - A United Object-Oriented Language for Physical Systems Modeling Language Specification (Version 2.2)

February 2, 2005
by the Modelica Association

注：Modelica®是 Modelica 协会的注册商标。

摘 要

本文详细说明 Modelica 建模语言(V2.2)，该语言由瑞典 Linköping 的非赢利组织 Modelica 协会开发，是一种适用于大规模复杂异构物理系统建模的面向对象语言，可以免费使用。Modelica 可以满足多领域建模需求，例如机电模型(机器人、汽车和航空应用中的机电系统包含了机械、电子、液压和控制子系统)、过程应用、电力发电和输送等。Modelica 模型的数学描述是微分、代数和离散方程组，相关的 Modelica 工具能够决定如何自动求解方程变量，因而无需手工处理。对具有超过 10 万个方程的大规模模型，可以使用专门的算法进行有效处理。Modelica 还适用于半实物仿真和嵌入式控制系统。更多关于 Modelica 的信息请浏览 **http://www.Modelica.org/**。

Abstract

This document defines the Modelica language, version 2.2, which is developed by the Modelica Association, a non-profit organization with seat in Linköping, Sweden. Modelica is a freely available, object-oriented language for modeling of large, complex, and heterogeneous physical systems. It is suited for multi-domain modeling, for example, mechatronic models in robotics, automotive and aerospace applications involving mechanical, electrical, hydraulic and control subsystems, process oriented applications and generation and distribution of electric power. Models in Modelica are mathematically described by differential, algebraic and discrete equations. No particular variable needs to be solved for manually. A Modelica tool will have enough information to decide that automatically. Modelica is designed such that available, specialized algorithms can be utilized to enable efficient handling of large models having more than one hundred thousand equations. Modelica is suited and used for hardware-in-the-loop simulations and for embedded control systems. More information is available at **http://www.Modelica.org/**.

目 录

1 引言(Introduction)

1.1 Modelica 概述(Overview of Modelica)

作为一种物理系统建模语言，Modelica 能有效支持模型库的开发和模型交换。Modelica 基于非因果建模思想，采用数学方程组和面向对象结构来促进模型知识的重用。

1.2 本规范的界定(Scope of specification)

Modelica 语言的语义通过一套模型翻译规则进行定义，这套规则解释如何将 Modelica 语言描述的模型转换成相应的平坦化混合 DAE 形式。模型翻译(可以理解为面向对象术语“实例化”)的关键在于：

- 继承基类展开。
- 基类、局部类和组件的参数化。
- 连接方程(来自 **connect** 语句)的生成。

平坦化 DAE 形式所描述的模型包括：

- 变量声明(带有基本类型、前缀和属性，例如 parameter Real v = 5)。
- 方程组(来自 equation 部分)。
- 函数调用(一次函数调用视为一组包括所有输入变量和结果变量的方程，方程数＝基本结果变量数)。
- 算法部分(每个算法被视为一组与算法中变量相关的方程，方程数＝不同的赋值变量数)。
- When 子句(每个 when 子句视为一组条件估值方程，也称瞬态方程，这些方程是该子句中变量的函数，方程数＝不同的赋值变量数)。

因此，可将混合 DAE 模型看成一组方程，其中一些方程是按条件计算的(例如瞬态方程，只有当相应的 when 条件转为 true 的瞬时才作计算)。

Modelica 语言规范不定义模型仿真结果，也不解释在数学上恰定的模型是如何构成的。

1.3 定义和术语(Definitions and glossary)

Modelica 语义规范应该和 Modelica 文法放在一起理解。非标准文本(即例子和注释)写在*[]*之中，用*[斜体]*书写。

术语	定义
组件	Modelica 文法中组件子句产生式定义的元素。
元素	类中声明的类定义、继承子句和组件子句。
实例化	将 Modelica 语言描述的模型转换成相应的混合 DAE 形式，包括继承基类展开，基类、局部类和组件的参数化，**connect** 语句连接方程的生成。

2 Modelica 语法(Modelica syntax)

2.1 词法约定(Lexical conventions)

采用以下扩展 BNF 句法符号：

[ ] —— 可选的
{ } —— 重复 0 或多次

定义如下词法单元：

IDENT = NONDIGIT { DIGIT | NONDIGIT } | Q-IDENT
Q-IDENT = "'" (Q-CHAR | S-ESCAPE) { Q-CHAR | S-ESCAPE } "'"
NONDIGIT = "_" | 小写字母"a"到"z" | 大写字母"A"到"Z"
STRING = """ { S-CHAR | S-ESCAPE } """
S-CHAR = 除双引号"""和反斜线"\"之外的任意字符
Q-CHAR = 除单引号"'"和反斜线"\"之外的任意字符
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]

[单引号是标识符的一部分，例如，'x' 和 x 是不同的标识符]

注意：字符串常量连接，例如，"a" "b"连接为"ab"(类似 C 语言)，在 Modelica 中用操作符"+"代替。

Modelica 采用 C++和 Java 一样的注释语法，也有注解和字符串注释形式的结构化注释。注释中"<HTML> …. </HTML>"序列表示 HTML 代码，可用于产生模型说明文档。

粗体表示 Modelica 语言的关键字。关键字是保留的，不能用作标识符(**initial** 除外)。**initial** 可以作为段落开头的关键字，也可能用作函数调用 **initial**()。

2.2 文法(Grammar)

2.2.1 存储定义(Stored definition)

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

2.2.2 类定义(Class definition)

class_definition :
[**encapsulated**]
[**partial**]
(**class** | **model** | **record** | **block** | [expandable] **connector** | **type** |
package | **function**)
class_specifier

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

base_prefix :
type_prefix

enum_list :
enumeration_literal { "," enumeration_literal }

enumeration_literal :
IDENT comment

composition :
element_list
{ **public** element_list |
protected element_list |
equation_clause |
algorithm_clause
}
[**external** [language_specification]
[external_function_call] [annotation] ";"〔原文为[annotation ";"]〕
[annotation ";"]]

language_specification :
STRING

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

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

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

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

2.2.3 继承(Extends)

extends_clause :
 extends name [class_modification] [annotation]

constraining_clause :
 extends name [class_modification]

2.2.4 组件子句(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 [conditional_attribute] comment

conditional_attribute :
 if expression

declaration :
 IDENT [array_subscripts] [modification]

2.2.5 变型(Modification)

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

class_modification :
"(" [argument_list] ")"

argument_list :
argument { "," argument }

argument :
  element_modification_or_replaceable
| element_redeclaration

element_modification_or_replaceable :
[**each**] [**final**] (element_modification | element_replaceable)

element_modification :
component_reference [modification] string_comment

element_redeclaration :
  redeclare [**each**] [**final**]
((class_definition | component_clause1) | element_replaceable)

element_replaceable :
replaceable (class_definition | component_clause1)
  [constraining_clause]

component_clause1 :
type_prefix type_specifier component_declaration1

component_declaration1 :
declaration comment

2.2.6 方程(Equations)

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

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

equation :
  (simple_expression "=" expression
| conditional_equation_e

| for_clause_e
| connect_clause
| when_clause_e
| IDENT function_call)
comment

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

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

conditional_equation_a :
if expression **then**
{ algorithm ";" }
{ **elseif** expression **then**
{ algorithm ";" }
}
[**else**
{ algorithm ";" }
]
end if

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

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

end for

for_indices :
for_index {"," for_index }

for_index :
IDENT [**in** expression]

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

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

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

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

2.2.7 表达式(Expressions)

expression :
simple_expression
| **if** expression **then** expression { **elseif** expression **then** expression } **else** expression

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

logical_expression :
logical_term { **or** logical_term }

logical_term :
logical_factor { **and** logical_factor }

logical_factor :
[**not**] relation

relation :
    arithmetic_expression [rel_op arithmetic_expression]

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

arithmetic_expression :
    [add_op] term { add_op term }

add_op :
    "+" | "-"

term :
    factor { mul_op factor }

mul_op :
    "*" | "/"

factor :
    primary ["^" primary]

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

name :
    IDENT ["." name]

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

function_call :
    "(" [function_arguments] ")"

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

named_arguments :
    named_argument ["," named_arguments]

named_argument :
    IDENT "=" expression

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

expression_list :
    expression { "," expression }

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

subscript :
    ":" | expression

comment :
    string_comment [annotation]

string_comment :
    [STRING { "+" STRING }]

annotation :
    annotation class_modification

3 Modelica 语义(Modelica semantics)

3.1 基本原则(Fundamentals)

实例化是在由环境和有序父集合组成的上下文中进行的。

3.1.1 作用域和名字查找(Scoping and name lookup)

3.1.1.1 父(Parents)

在词法上，封装某个元素的一系列类形成有序父的集合。一个类 B 定义于另一个类 A 之中，则在 B 的元素的有序父集合中，B 位于 A 的前面。*[规范原文中没有标识符 A、B，为准确解释有序父集合中的元素顺序而特意添加。]*

未命名父封装所有顶层类定义(及其所有子类)，以及 3.1.4 节描述的尚未读入的类定义。在未命名父中，顶层类定义的顺序未作明确规定。

实例化期间，对于某个正在实例化的元素来说，其父是一个部分实例化的类。*[这意味着一个声明能够引用 extends 子句中继承而来的名字。]*

[例：

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

类定义 C3 的未命名父以任意顺序包含 C1、C2 和 C3。当实例化类定义 C3 时，声明 x 的父集合是部分实例化的类 C3 以及包含 C1、C2 和 C3 的未命名父。元素 z 的父集合依次是 C4、C3 和未命名父。]

3.1.1.2 静态名字查找(Static name lookup)

类实例化时，名字查找目的是找到基类、组件类型等元素的名字。类型名字查找期间，记录构造函数隐式定义的名字被忽略*[参见 3.4.8.3 节，记录与隐式创建的记录构造函数名字相同]*。在函数名查找期间，记录类名也被忽略。

对于简单名字*[未使用"."符号组合]*，查找过程如下：

- 如果在 for 循环体(3.3.3 节)、或约简表达式中(3.4.3.1 节)，首先查找隐式声明的迭代变量。
- 实例化元素*[元素指嵌套类、继承子句或组件子句]*、方程或算法时，任何名字的查找都是顺序检查其有序父集合中的每个成员，直到找到一个匹配、或父被封装为止。后者情况下*[指父被封装]*，除非是查找本规范预定义的类型、函数和操作符，查找过程停止。这些情况下*[指查找预定义元素]*，查找继续在他们定义的全局范围内继续进行。*[例如，在层次结构中照常向上搜索 abs, 如果到达某个封装边界，就在全局范围内进行搜索。操作符 abs 不能在全局作用域内被重定义，原因是已经存在的类不能在同一层次中重新定义。]*对于封装父类中能够查找到的变量，只有当变量声明为常量(**constant**)时才允许使用。3.2.7 节介绍了在简短类定义的变型中使用情况。

  [例：

  ```
  package A
      constant Real x = 2;
      model B
          Real x;
          function foo
              output Real y
          algorithm y := x; //非法引用 B 的非 constant 变量 x。]
  ```

- 每个作用域中的查找如下进行：

  1. 在该类声明的命名元素(类定义和组件声明)中查找(包括从基类继承的元素)。
  2. 在词法作用域中限定性导入语句所导入的名字中查找。语句"import A.B.C;"导入的名字是 C，语句"import D=A.B.C;"导入的名字是 D。
  3. 在词法作用域内非限定性导入语句所导入的包的公有成员中查找。如果这一步在多个非限定性导入(包)中产生匹配，则报错。

[注意：定义于继承类中的导入语句在查找时忽略，即导入语句不被继承。]

对于形如 A.B(或 A.B.C 等)的复合名字，查找过程如下：

- 按上述简单名字的方式查找第一个标识符*[A]*。
- 如果第一个标识符表示某个组件，则该名字的其余部分(例如 B 或 B.C)在该组件声明的命名组件元素中查找。
- 如果第一个标识符表示类，那么该类首先使用其父、在空环境下(即没有变型，参见 3.1.2 节)进行临时实例化。随后，复合名字的其余部分(例如 B 或 B.C)就在临时实例化类声明的命名元素中查找。如果这个(临时实例化)类不满足包的要求，那么查找限制为只针对被封装的元素。

[对复合名字进行的临时类实例化，遵循与继承子句的基类、局部类、组件子句的类型实例化相同的规则，只是这里环境为空。]

查找导入的包或类的名字时，例如语句 **import** A.B.C;、**import** D=A.B.C;、**import** A.B.C.*中的 A.B.C，与正常的词法查找不同，这种查找从顶层开始按词法方式查找名字的第一部分。

限定性导入语句只能引用包或包的元素，即在"import A.B.C;"或"import D=A.B.C"语句中，A.B 必须是一个包。非限定性导入语句只能从包中导入，即在"import A.B.*;"中 A.B 必须是一个包。*[注意：语句"import A;"中 A 可以作为未命名顶层包元素的任意类。]*

3.1.1.3 动态名字查找(Dynamic name lookup)

以前缀 **outer** 声明的元素所引用的元素实例是使用前缀 **inner** 声明的同名元素，该实例最接近于封装 **outer** 元素声明的实例层次中。

对一个 **outer** 元素引用来说，应至少存在一个对应的 **inner** 元素声明。*[inner/outer 组件可用于简单场的建模，有一些物理量(例如重力矢量、环境温度或环境压力)可在特定的模型层次中被所有组件访问。如果 inner 组件未被模型结构嵌套层次中对应的非 inner 声明所屏蔽，则他们可以在整个模型中被访问到。]*

[简单例子：

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

class B
    inner Real T0;
    A a1, a2; //B.T0、B.a1.T0 和 B.a2.T0 是同一个变量
    ...
end B;
```

更复杂的例子：

```
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 和 e.f.TI 是同一个变量
    //但是 e.f.TI、e.TI 和 TI 是不同的变量
    A a; //a.TI、a.b.c.d.TI 和 TI 是同一个变量
end I;
```

]

inner 组件应作为对应 outer 组件的子类型。*[如果这两个类型不一致，对 inner 组件类型所定义的实例来说，outer 组件只引用了 inner 组件的一部分。]*

[例：

```
class A
    inner Real TI;
    class B
        outer Integer TI; //错误，因为 A.TI 不是 A.B.TI 的子类型
    end B;
end A;
```

inner 声明能用于定义场函数，如与位置有关的重力场，例：

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

function B //B 是 A 的子类型
    extends A;
algorithm
    ...
end B;

class C
    inner function fc = B; //定义实际使用的函数
    class D
        outer function fc = A;
            ...
        equation
            y = fc(u);   //使用函数 B。
    end D;
end C;
```

]

同时用前缀 inner 和 outer 声明的元素从概念上引入了相同名字的两个声明：一个遵循上述 inner 规则，另一个遵循 outer 规则。*[对于带有 inner 和 outer 前缀的元素，其局部引用的是 outer 元素，即在前缀 inner 封装的作用域内依次引用对应的元素。]*

只有当 outer 元素声明同时带有前缀 inner 时，才可以对其进行变型*[包括声明方程]*。而变型只作用于 inner 声明。

[例：

```
class A
    outer parameter Real p=2; //错误，因为[只有 outer 时不能]变型
end A;
```

在下面的例子中：isEnabled[原文为 enabled]在模型层次中传播，也能在局部抑制 subSystem[原文为 subsystems]。

```
model ConditionalIntegrator "Simple differential equation if isEnabled"
    outer Boolean isEnabled;
    Real x(start=1);
    equation
        der(x) = if isEnabled then –x else 0;
end ConditionalIntegrator;

model SubSystem "subsystem that 'enable' its conditional integrators"
```

```
    Boolean enableMe = time <= 1;
    //设置 inner isEnabled 等于((outer isEnabled) and enableMe)
    inner outer Boolean isEnabled = isEnabled and enableMe;
    ConditionalIntegrator conditionalIntegrator;
    ConditionalIntegrator conditionalIntegrator2;
end SubSystem;

model System
    SubSystem subSystem;
    inner Boolean isEnabled = time >= 0.5;
    //subSystem.conditionalIntegrator.isEnabled 等于
    //'isEnabled and subSystem.enableMe'
end System;
```

]

3.1.2 环境和变型(Environment and modification)

3.1.2.1 环境(Environment)

环境包含修改类元素(例如 parameter 变化)的参数。环境通过合并类的变型而建立，其中外层变型覆盖内层变型。

3.1.2.2 变型合并(Merging of modifications)

变型合并意味着外层变型覆盖内层变型。合并是有层次的，整个非简单类型的变型值覆盖所有组件的值变型，但如果覆盖组件的 final 属性则是错误的。合并变型时，每个变型保持其自身的 **each** 属性。

[下面一个较大的例子说明了变型的几个方面问题：

```
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);      //正确，C11 已从 C1 继承
    C21 u;             //正确，尽管 C21 是在下面声明从 C2 继承的
    extends C2;
end C3;
```

元素 t 声明的环境是(x=3)。环境通过合并类变型而形成，如下所示：

```
class C1
    parameter Real a;
end C1;

class C2
    parameter Real b, c;
end C2;

class C3
    parameter Real x1;          //无缺省值
    parameter Real x2 = 2;      //缺省值为 2
    parameter C1 x3;            //x3.a 无缺省值
    parameter C2 x4(b = 4);     //x4.b 缺省值为 4
    parameter C1 x5(a = 5);     //x5.a 缺省值为 5
    extends C1;                 //继承而来的元素 a 没有缺省值
    extends C2(b = 6, c = 77);  //继承得到的元素 b 缺省值为 6
end C3;

class C4
    extends C3(x2 = 22, x3(a = 33), x4(c = 44), x5 = x3, a = 55, b = 66);
end C4;
```

*外层变型覆盖内层变型，例如*b=66 *覆盖了*extends C2(b=6)*中嵌套的类变型。这就是变型合并：合并*((b=66),(b=6))*使得*(b=66)。

类 C4 的实例化给出具有下列变量的对象：

变量	*缺省值*
x1	*None*
x2	*22*
x3. a	*33*
x4. b	*4*
x4. c	*44*
x5. a	*x3. a*
a	*55*
b	*66*
c	*77*

]

3.1.2.3 单一变型(Single modification)

一个变型的两个参数不能对元素的同一个原生属性赋值。当使用限定性名字时，以相同标识符开头的不同限定性名字被合并到一个变型。

[例：

```
class C1
    Real x[3];
end C1;
class C2 = C1(x = ones(3), x[2] = 2); //错误：x[2]被赋值两次
class C3
    class C4
        Real x;
    end C4;
    C4 a(x.unit = "V", x.displayUnit = "mV", x = 5.0);
    //正确，赋值的属性不同(unit、displayUnit 和 value)
    //等同于
    C4 b(x(unit = "V", displayUnit = "mV") = 5.0));
end C3;
```

]

3.1.2.4 实例化顺序(Instantiation order)

声明元素的名字不应与其部分实例化父类中任何其他元素同名。组件不应与其类型标识符同名。

变量和类可以在声明之前使用。

[实际上，(元素)声明顺序只对以下对象有意义：

- *有多个输入变量、使用位置参数调用的 function，参见 3.4.8 节。*
- *有多个输出变量的 function，参见 3.4.8 节。*
- *用于外部函数参数的 record，参见 6.2.3 节。*
- *enumeration 类型中枚举常量的顺序，参见 3.2.3.1 节。]*

为保证元素能够在声明之前使用，以及元素不依赖在父类中的声明顺序，实例化过程应按以下步骤进行：

1．平坦化

首先，找到声明的局部类和组件的名字。这里，变型被合并到局部元素，重声明生效。然后查找基类，将其平坦化并插入到(当前)类中。基类的查找应该独

立进行[继承类名字的查找应该在继承子句平坦化前后给出相同的结果。平坦化期间，所使用的任何元素不应通过查找继承子句得到。在插入平坦化结果之前应展开类中所有的继承子句。在插入继承子句平坦化结果之前，应展开用于继承的局部类。]

2．实例化

平坦化类，应用变型，实例化所有的局部元素。

3．平坦化检查

检查重复的元素*[源于多继承]*在实例化之后是否一致。

4．数组元素变型

变型关键字 **each** 用于数组声明或变型时，该变型单独作用于数组中每一个元素。如果变型元素是一个向量，而且变型不包含 **each** 属性，则该变型被分裂，使得变型向量中第一个元素作用于元素向量的第一个元素，第二个作用于第二个元素，等等。矩阵和一般元素数组被视为向量的向量进行处理。

如果变型元素是带下标的向量，则下标必须是 Integer 常量。

如果嵌套的变型被分裂，则这种分裂传播到嵌套变型的所有元素。但如果变型带有 each 属性，则针对这些变型元素的分裂被抑制。如果按这种方式分裂的嵌套变型中含有重声明，则这种分裂是非法的。

[例：

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

这意味着 c[i].a[j]=j，c[i].d=i。*]*

3.1.3 子类型和类型等价(Subtyping and type equivalence)

3.1.3.1 类的子类型(Subtyping of classes)

对于任何类 S 和 C，如果他们是等价的或满足下列条件，则 S 是 C 的超类型，C 是 S 的子类型：

- S 的每个公有声明元素也存在于 C 中(按名字)
- S 中这些元素的类型是 C 中相应元素类型的超类型。

基类是在继承子句中引用的类。包含继承子句的类称为导出类。*[C 的基类通*

常是 C 的超类型，但是其他没有与 C 有继承关系的类也可能是 C 的超类型。]

3.1.3.2 组件的子类型(Subtyping of components)

如果满足下列条件，则组件 B 是 A 的子类型：

- 二者同为标量或具有相同维度的数组。
- B 的类型是 A 的基类型(数组基类型)的子类型。
- 对于数组的每一维：
  - A 的大小不确定，或
  - 表达式“(B 的大小)－(A 的大小)”的值恒等于 0(在 B 的环境中)。

3.1.3.3 类型等价(Type equivalence)

类型 T 和 U 满足下列条件时称二者是等价的：

- T 和 U 表示相同的内置类型(RealType、IntegerType、StringType、BooleanType)，或
- T 和 U 均是类，T 和 U 包含相同的公有声明元素(按名字)，T 的元素类型与 U 对应的元素类型是等价的。

3.1.3.4 类型一致(Type identity)

元素 T 和 U 满足下列条件时称二者是一致的：

- T 和 U 是等价的。
- 二者要么都声明为 final，要么都不为 final。
- 如果 T 和 U 是组件，要求他们的类型前缀(参见 3.2.1 节)是一致的，并且
- 如果 T 和 U 是类，T 和 U 含有相同的公有声明元素(按名字)，T 的元素类型与 U 对应的元素类型是一致的。

3.1.3.5 有序类型一致(Ordered type identity)

当且仅当满足下列条件时，元素 T 和 U 是有序类型一致的：

- T 和 U 是类型一致的。
- 如果T 和 U 是类：
  - T 和 U 的元素数目相同。

- T 的第 i 个声明元素与 U 的第 i 个声明元素是有序类型一致的。

3.1.3.6 函数类型一致(Function type identity)

当且仅当满足下列条件时，函数 T 和 U 是类型一致的：

- T 和 U 有相同数目的输入和输出元素。
- 对每个输入或输出元素：
  - 对应元素的名字相同。
  - 对应元素是有序类型一致的。

3.1.3.7 枚举类型等价(Enumeration type equivalence)

枚举类型 S 和 E 满足下列条件时称二者是等价的：

- S 和 E 具有相同数目的枚举常量。
- S 的第 i 个枚举常数与 E 的第 i 个枚举常量名字相同。

3.1.3.8 枚举类型的子类型(Subtyping of Enumeration types)

对于任何枚举类型 S 和 E，如果 他们是等价的或 S 是 enumeration(:)类型，则 S 是 E 的超类型，E 是 S 的子类型。

[例：

```
E1 = enumeration(one,two,three);          //E1 不是 E2 的子类型
E2 = enumeration(one,two,three,four);     //E2 不是 E1 的子类型
E3 = enumeration(one,two,three,four);     //E3 是 E2 的子类型
E4 = enumeration(:);                      //E1、E2、E3 是 E4 的子类型
```

]

3.1.4 类的外部表示(External representation of classes)

类可用操作系统*[文件系统或数据库]*的层次结构进行表示。带有版本信息的类参见 7.4.3 节的说明。这种外部实体分为以下两种形式：

- 结构化实体*[例如文件系统的一个目录]*。
- 非结构化实体*[例如文件系统的一个文件]*。

3.1.4.1 结构化实体(Structured entities)

结构化实体*[例如目录 A]*应包含一个结点。在文件层次中，该结点存储于package.mo 文件。该结点应包含定义类*[A]*的存储定义(stored-definition)，且类名与结构化实体的名字匹配。*[该结点通常包含 package 的文档和图示信息，但也可能包含类 A 的其它元素。]*

结构化实体可能包含一个或多个子实体(结构化或非结构化的)。子实体映射为封装的结构化实体所定义的类元素。*[例如，目录 A 包含 3 个文件：package.mo、B.mo 和 C.mo，定义的类分别为 A、A.B 和 A.C。]*两个子实体不能定义同名的类*[例如，一个目录不能同时包含子目录 A 和文件 A.mo]*。

3.1.4.2 非结构化实体(Non-structured entities)

非结构化实体*[例如文件 A.mo]*应只包含一个模型定义(model-definition)来定义类*[A]*，类名与非结构实体的名字匹配。

3.1.4.3 Within 子句(Within clause)

非顶层实体应以 within 子句开始，描述实体中定义的类在 Modelica 类层次中的位置。顶层类可以包含一个没有名字的 within 子句。

对于封装的结构化实体中的子实体，within 子句应指明封装实体的类。

3.1.4.4 使用 MODELICAPATH(Use of MODELICAPATH)

顶层作用域隐含包含大量外部存储的类。如果某个顶层名字在全局作用域中找不到，Modelica 翻译器应该在一个有序的库根目录列表(称为 MODELICAPATH)中查找另外的类。*[在典型系统中，MODELICAPATH 是一个环境变量，包含由分号分隔的目录名列表。]*

[路径 A.B.C 的第一部分(即 A)通过搜索 MODELICAPATH 的有序根目录列表来定位。如果没有根目录包含 A，则查找失败。如果 A 在某个根目录中被找到，则路径的其余部分定位于 A 中；如果定位(A)失败，则整个查找失败，无需再在 MODELICAPAT 的其它根目录中查找 A。]

3.2 声明(Declarations)

3.2.1 组件子句(Component clause)

如果组件的类型标识符表示某个内置类型(RealType、IntegerType 等等)，则

实例化的组件具有相同的类型。

如果组件的类型标识符不是内置类型，就需要查找类型名字(3.1.1 节)。找到的类型在新的*[变型]*环境和该组件的部分实例化父类中被实例化。如果类型是抽象的(partial)，则出现错误。新的环境是顺序合并以下变型的结果：

- 父的元素变型中与该组件同名的变型
- 组件声明的变型

定义了内置类型组件数值的环境也可以说是定义了一个与该声明组件关联的声明方程。对于向量和矩阵声明，声明方程关联到每个元素。*[这使得有可能在父变型元素中重写单个元素的声明方程，另外，不可能将声明方程视为单个的矩阵方程。]*

数组维度应该是非负的参数表达式，或是冒号操作符，表示数组维度待定。

声明有类型前缀 **flow** 的变量应为 Real 的子类型。

类型前缀(即 flow、discrete、parameter、constant、input、output)只适用于 type、record 和 connector 组件。复合组件的类型前缀 flow、input、output 也作用于组件的元素。只有当组件的元素都没有对应的 flow、input、output 类型前缀时，类型前缀 flow、input、output 才能用于复合组件。*[例如，只有在没有元素具有 input 或output 类型前缀时，input 才能用于(组件)]*。类型前缀 discrete、parameter、constant 的相应规则在 3.2.2.1 节描述。

对于函数类型的组件和函数中的组件，相应的规则在 3.2.13 节描述。

条件组件声明在 3.2.14 节描述。

3.2.2 可变性前缀(Variability prefix)

组件声明前缀 **discrete**、**parameter**、**constant** 称为可变性前缀，定义了组件变量值在什么环境中被初始化(参见 3.5 节)，以及在瞬态分析中什么时候被改变(即混合 DAE 的初值问题求解)：

- 用前缀 **parameter** 或 **constant** 声明的变量 vc 在瞬态分析中保持不变。
- 离散时间(**discrete**-time)变量 vd 的时间导数等于 0(非正式地，der(vd)=0，但是将操作符 der()用于离散时间变量是非法的)，在瞬态分析期间只能在事件触发时刻改变其变量值(参见 3.5 节)。
- 连续时间(continuous-time)变量 vn 时间导数可以不为 0(der(vn)≠0)，可在瞬态分析中随时改变其变量值。

用前缀 **discrete** 声明的 Real 变量必须在 when 子句中通过赋值语句或方程进行赋值。

即使未用前缀 **discrete** 声明，在 when 子句中赋值的 Real 变量也是离散时间变量。不在任何 when 子句中赋值而且没有任何类型前缀的 Real 变量是连续时间变量。

Integer、String、Boolean 或 enumeration 变量的缺省可变性是离散时变的，不能声明连续时变的 Integer、String、Boolean 或 enumeration 变量。*[因为有作用于离散表达式的限制，Modelica 翻译器能够保证该特性，参见 3.4.9。]*

表达式的可变性以及定义方程的可变性限制在 3.4.9 节给出。

*[离散时间(**discrete**-time)变量是一种分段常量信号，在仿真期间，仅在事件触发时改变变量值。这种变量类型在应用一些特定算法时是需要的，例如用于指标约简的 Pantelides 算法，必须知道这些变量的时间导数何时等于 0。进一步地，如果已知组件只在事件触发时发生变化，就可以降低仿真环境中的内存需求。*

*参数(**parameter**)变量在仿真期间保持不变。前缀 parameter 允许库的开发者表示这样一种意图，只有当一些用到的组件在仿真期间保持不变时，库中的物理方程才是有效的。离散时间(**discrete**-time)变量和常量(**constant**)变量也是如此。另外，前缀 **parameter** 允许在试验环境中采用方便的图形用户界面支持快速修改已编译模型中最重要的常量。前缀 **parameter** 与 if 子句结合，支持在进行模型符号处理之前移除模型的部分内容，以消除模型中变量的因果性质(类似于 C 语言的 #ifdef)。类参数有时也作为可选项使用。*

例：

```
model Inertia
    parameter Boolean state = true;
        ...
equation
    J * a = t1 – t2;
    if state then  //如果 state = false，在符号处理中这部分代码被移除
        der(v) = a;
        der(r) = v;
    end if;
end Inertia;
```

*常量(**constant**)变量与参数变量类似，不同的是，常量在给定值之后不能改变。常量可用于表示数学常数，例如：*

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

没有连续时变的 Boolean、Integer 或 String 变量，在极少数情况下需要时可以用 Real 变量代替，例如：

```
    Boolean off1, off1a;
    Real off2;
equation
    off1  = s1 < 0;
    off1a = noEvent(s1 < 0);                  //错误，因为 off1a 是离散的
    off2  = if noEvent(s2 < 0) then 1 else 0; //合理的
    u1 = if off1 then s1 else 0;              //生成状态事件
    u2 = if noEvent(off2 > 0.5) then s2 else 0; //无状态事件
```

因为 off1 是离散时间变量，产生的状态事件使得 off1 只在事件触发时改变。变量 off2 可以在连续积分期间改变变量值。这样，在连续积分期间 u1 确保是连续的，但对于 u2 不存在这种保证。]

3.2.2.1 复合实体的可变性(Variability of structured entities)

对于带有可变性前缀的复合实体，其元素的可变性取决于该元素的可变性前缀与组件的可变性中限制性较高的一项(如果组件没有可变性前缀，则使用组件的缺省可变性)。

[例：

```
record A
    constant Real pi = 3.14;
    Real y;
    Integer i;
end A;
parameter A a;
//a.pi 是常量
//a.y 和 a.i 是参数变量
A b;
//b.pi 是常量
//b.y 是连续时间变量
//b.i 是离散时间变量
```

]

3.2.3 参数绑定(Parameter binding)

在翻译后的模型中，参数和常量相关的声明方程在实例化之后必须是非循环的。因此不能通过循环依赖引入参数方程。

[例：

```
constant Real p = 2 * q;
constant Real q = sin(p); //非法，因为 p=2*q, q=sin(p)循环赋值

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

final ABCD a;              //非法，因为 size(a.A, 1)和 a.n 循环依赖

ABCD b(redeclare Real A[2, 2] = [1, 2; 3, 4]);
                           //合法，因为 A 的大小不依赖 n
```

ABCD c(n = 2);　　//合法，因为 n 不依赖 A 的大小。

]

3.2.4 保护元素(Protected element)

保护元素不能通过点号"."访问，不能在类变型中修改或重声明。

在标题 **protected** 下面定义的所有元素视为保护的。其它所有元素*[即在标题* ***public*** *下面定义的元素、没有标题的或在一个单独的文件中定义的]*是公有的*[即非保护的]*。

如果继承子句出现在标题 **protected** 下面，则基类所有的元素被视为当前类的保护元素。如果继承子句是公有的，则基类的所有元素以其自身的保护属性被继承*[即基类的公有元素成为派生类的公有元素，基类的保护元素成为派生类的保护元素]*。出现在基类最后的标题 protected 和 public 不影响当前类中后续的元素(即标题 **protected** 和 **public** 不被继承)。

3.2.5 数组声明(Array declarations)

Modelica 类型系统包括标量、向量、矩阵(维数 ndim=2)和超过 2 维的数组。*[行向量和列向量没有区别。]*

下表列出了两种可能的(数组)声明形式，并定义了相关术语。C 表示任意的类，包括内置类型 Real、Integer、Boolean、String 和枚举类型。数组某一维的上界表达式，例如下表中的 n、m、p 等，其类型要求是 Integer 的子类型、或者是枚举类型 E 的名字 E、或者是 Boolean。冒号(：)表示该维的上界是未知的，并且(类型)是 Integer 的子类型。如果维数下标类型是 Integer 的子类型，该维的下界为 1；如果维数下标类型是枚举类型 E=enumeration(e1,...,en)，该维的下界为 E.e1；如果维数下标类型是 Boolean 类型，该维的下界为 false。如果维数下标类型是枚举类型 E=enumeration(e1,...,en)，该维的上界为 E.en；如果维数下标类型是 Boolean 类型，该维的上界为 true。

数组下标为 Boolean 或枚举类型时，只能按下列方式使用：

- 使用相应类型(即 Boolean 或枚举类型)的表达式作为下标
- 对于数组，允许形如 x1 = x2 的声明方程以及形如 x1 := x2 的声明赋值，不管各维的下标类型是否为 Integer 的子类型、Boolean 或枚举类型。

Modelica 形式 1	Modelica 形式 2	#维数	含义	说明
C x;	C x;	0	标量	标量
C[n] x;	C x[n];	1	向量	n 维向量
C[E] x;	C x[E]	1	向量	下标为枚举类型 E 的向量

C[n, m] x;	C x[n, m];	2	矩阵	n×m 矩阵
C[n, m, p, ...] x;	C x[m, n, p ,...];	k	数组	k 维数组(k>=0)

[维数和各维大小是类型的一部分，应该对其进行检查(例如在重声明时)。声明形式 1 清楚地列出了数组的类型，而形式 2 是诸如 FORTRAN、C、C++等语言中数组声明的惯用方式。

Real[:] v1, v2;　　//向量 v1 和 v2 各维的大小不定，实际大小可能不同

混用两种声明形式是允许的，但不推荐这样表示。

Real[3,2] x[4,5];　//x 的类型是 Real[4,5,3,2];

以枚举值为下标的向量 y，

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

]

零维数组是允许的，"C x[0];"声明了一个空向量，而"C x[0,3];"声明了一个空矩阵。

[特例：

Modelica 形式 1	Modelica 形式 2	#维数	含义	说明
C[1] x;	C x[1];	1	向量	1 维向量，表示一个标量
C[1,1] x;	C x[1,1];	2	矩阵	1×1 矩阵，表示一个标量
C[n,1] x;	C x[n,1];	2	矩阵	n×1 矩阵，表示一列
C[1,n] x;	C x[1,n];	2	矩阵	1×n 矩阵，表示一行

]

“数组的数组”类型是多维数组，多维数组的构造方式如下：开头的维数取自组件声明，后面的维数取自作“完全张开(maximally expanded)”的组件类型。如果某个类型是内置类型之一(Real、Integer、Boolean、String、枚举类型)，或者不是 type 类，那么该类型是完全张开的。在操作符重载之前，变量的 type 类是完全张开的。

[例：

type Voltage = Real(unit = “V”);
type Current = Real(unit = “A“);
connector Pin
　Voltage　　v;　　//v 的 type 类是 Voltage，v 的类型是 Real
　flow Current i;　　//i 的 type 类是 Current，i 的类型是 Real
end Pin;

```
type MultiPin = Pin[5];

MultiPin[4] p;          //p 的 type 类是 MultiPin，p 的类型是 Pin[4,5];
type Point = Real[3];
Point  p1[10];
Real   p2[10,3];
```

组件 p1 和 p2 的类型是一致的。

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

]

[关于仿真时的自动断言：

设 A 为声明的数组，i 为声明的 di 维的最大维度。如果在编译时没有检查到断言“assert(i>=0, ...)”，应生成该断言语句。有一个实现上的质量问题：如果断言失败，应产生一条友好的错误信息。

设 A 为声明的数组，i 为访问 di 维一个索引的下标。对于每次下标访问，如果在编译时没有检查到断言“assert(i>=1 and i<=size(A,di), ...)”，应生成该断言语句。

考虑效率的原因，可以选择性地抑制这些隐式断言语句。]

3.2.6 Final 元素变型(Final element modification)

在元素变型或声明中定义为 **final** 的元素不能被变型或重声明修改。**final** 元素的所有元素也是 **final**。*[在试验环境中设置一个参数值在概念上视为一次变型。这意味着参数的 final 变型方程不能在仿真环境中被修改。]*

[例：

```
type Angle = Real(final quantity=”Angle”, final unit =”rad”,
                                          displayUnit=”deg”);
Angle a1(unit=”deg”);           //错误，因为 unit 声明为 final!
Angle a2(displayUnit=”rad”);    //正确

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);       //正确
    PI c2(b={1});          //错误，b 声明为 final
  end Test;
```

]

3.2.7 简短类定义(Short class definition)

类定义形式

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

与较长一点的类定义

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

是一致的，除了二者在变型的词法作用域上不一样，简短类定义没有为变型引入额外的词法作用域。

[例：说明变型不同的作用域

```
  model Resistor
    parameter Real R;
    ...
  end Resistor;
  model A
    parameter Real R;
    replaceable model Load=Resistor(R=R) extends TwoPin;
    //正确，设置 Resistor 的 R 等于模型 A 的 R。
    replaceable model LoadError
      extends Resistor(R=R);
      //这里给出了一个奇异方程 R=R，因为右边的 R
      //既能在 LoadError 中找到，又能在其基类 Resistor 中找到。
    end LoadError extends TwoPin;
    Load a,b,c;
    ConstantSource ...;
    ....
  end A;
```

]

简短类定义形式如下：

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

这里 N 表示任意的数组维数，在概念上视为一个数组类：

‘array’ TN
    T[N] _ (optional modifiers);
’end’ TN;

这样的数组类具有确定的匿名组件(_)。这种数组类型的组件实例化时，产生的实例化组件类型是一个数组类型，与(_)具有相同的维数，并带有可选的变型。

[例：

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

其中，f1 和 f2 的类型是一致的。]

如果简短类从抽象类继承，则新的类定义也是抽象类，不管该类是否用了关键字 partial 进行声明。

[例：

replaceable model Load=TwoPin;
Load R; //错误，因为 TwoPin 是一个抽象类，除非 Load 被重声明。

]

如果简短类定义没有声明为任何受限类，新的类定义将继承该受限类(该规则反复作用，对重声明也是如此)。

在简短类定义中，基本前缀不影响类型本身，但会作用于该类型或其派生类型所声明的组件。对于从数组类、带有基本前缀的类或简单类型继承的组件，组合其基本前缀是非法的。

[例：

type InArgument=input Real;
type OutArgument=output Real[3];
function foo
    InArgument u; //等价于‘input Real u’
    OutArgument y; //等价于‘output Real[3] y’
algorithm
    y:=fill(u,3);
end foo;
Real x[:]=foo(time);

]

3.2.7.1 枚举类型(Enumeration types)

形如

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

的声明定义枚举类型 E 及其相关的枚举常量列表 enum_list。这是关键字 **enumeration** 唯一合法的用法。枚举类型的枚举常量应互不相同。枚举常量的名字在 E 的作用域内定义。enum_list 中每个枚举常量的类型都为 E。

[例：

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

]

每个枚举常量可带有一个可选的注释字符串：

[例：

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

]

枚举类型是简单类型，其属性定义在 3.6 节。布尔类型的名字或枚举类型的名字可以用于数组声明，指定数组维数的界限，也可用于 for 循环的范围表达式，指定循环范围。枚举类型元素可以在表达式中访问[例如数组下标值]。

[例：

type DigitalCurrentChoices = **enumeration**(zero, one);
//类似 Real、Integer

设置属性：

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

在表达式中访问属性值：

Real x[DigitalCurrentChoices];

//用类型名表示循环范围
for e **in** DigitalCurrentChoices **loop**
x[e] := 0.;
end for*[原文为* ***end loop****]*;

for e **loop** //等价的例子，其中使用更简洁的形式

```
    x[e] := 0.;
end for[原文为 end loop];

//等价的例子，使用冒号范围构造器
for e in DigitalCurrentChoices.zero : DigitalCurrentChoices.one loop
    x[e] := 0.;
end for[原文为 end loop];

model Mixing1 ”Mixing of multi-substance flows, alternative 1”
    replaceable type E=enumeration(:) ”Substances in Fluid”;
    input    Real c1[E], c2[E], mdot1, mdot2;
    output Real c3[E], mdot3;
equation
    0 = mdot1 + mdot2 + mdot3;
    for e in E loop
        0 = mdot1*c1[e] + mdot2*c2[e] + mdot3*c3[e];
    end for;
    /*  不能对枚举类型使用数组运算：
        zeros(n) = mdot1*c1 + mdot2*c2 + mdot3*c3 //错误
    */
end Mixing1;

model Mixing2 ”Mixing of multi-substance flows, alternative 2”
    replaceable type E=enumeration(:) ”Substances in Fluid”;
    input    Real c1[E], c2[E], mdot1, mdot2;
    output Real c3[E], mdot3;
protected
    //没有效率损失，因为 cc1, cc2, cc3 可以在翻译期间去除
    Real cc1[:]=c1, cc2[:]=c2, cc3[:]=c3;
    final parameter Integer n = size(cc1,1);
equation
    0 = mdot1 + mdot2 + mdot3;
    zeros(n) = mdot1*cc1 + mdot2*cc2 + mdot3*cc3
end Mixing2;
```

]

3.2.7.2 函数偏导数(Partial derivatives of functions)

形如

IDENT "=" **der** "(" name "," IDENT { "," IDENT } ")" comment

的类定义函数偏导数，这种方式只能用于函数声明。

上述语义表示，使用这种形式声明的函数*[而且只能是函数]*定义了等号右边函数的偏导数(查找方式与简短类定义相同——查找的名字必须是函数)，并且对每个 IDENT(从第一个开始)依次求偏微分。IDENT 必须是函数的 Real 类型输入。

允许使用 comment 允许对该函数作注释(处于 info 层，类似单行注释或者图标)。

[例：如下所示的 Gibbs 函数可用于计算特定的热焓：

```
function Gibbs
    input   Real p, T;
    output Real g;
algorithm
    ...
end Gibbs;
function Gibbs_T = der(Gibbs, T);
function specificEnthalpy
    input   Real p, T;
    output Real h;
algorithm
    h := Gibbs(p, T) – T * Gibbs_T(p, T);
end specificEnthalpy;
```

]

3.2.8 局部类定义(Local class definition)

局部类在其部分实例化父中进行静态实例化，抽象类或 outer 局部类的组件除外。*[实例化]*环境是与局部类同名的父类元素变型，或者是空环境。

未实例化的局部类连同其环境成为已实例化父类的元素。

[下例说明局部类参数化：

```
class C1
    class Voltage = Real(nominal=1);
    Voltage v1, v2;
end C1;
class C2
    extends C1(Voltage(nominal=1000));
end C2;
```

类 C2 的实例化产生带有 nominal 变型值为 1000 的局部类 Voltage。变量 v1 和 v2 实例化该局部类，因而他们的 nominal 值均为 1000。]

3.2.9 继承子句(Extends clause)

基类的名字在继承子句的部分实例化父中查找*[规则#1]*。找到的基类在一个新环境和继承子句的部分实例化父中实例化*[#2]*。新的环境是依次合并以下两项的结果*[#3]*：

1. 所有父的(变型)环境中与实例化基类名字匹配的参数。
2. 继承子句中可选的类变型。

[下面给出这三个规则的例子：

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

]

实例化基类的元素成为实例化父类的元素。

[从上面的例子中得到以下实例化的类：

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

给定以上定义的类A 和类B，合并规则的顺序确保得到下面的结果，

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

产生一个实例，其中bcomp.b=3，覆盖了b=2。]

实例化基类的声明元素应满足以下两者之一：

- 在部分实例化父类还不存在*[即具有不同的名字]*。
- 与实例化父类中的某一元素完全一致，即有相同的名字，相同的保护级别(**public** 或 **protected**)，和相同的内容。这种情况下，忽略其中的一个元素(因为他们是一致的，忽略哪一个没有关系)。

若不是以上两种情况之一，那么模型是错误的。

实例化基类中与实例化父类在语法上等价的方程将被丢弃。*[注意：数学上等价、但语法上不等价的方程不会丢弃，由此产生一种超定的方程系统。]*

3.2.10 重声明(Redeclaration)

变型中的 **redeclare** 结构使用另一个声明替换变型元素中局部类或组件的声明。作为一个独立元素时，redeclare 结构使用另一个声明替换继承的局部类或组件声明。

具有关键字 replaceable 的变型自动被看成一个重声明。

重声明中，原有声明的一部分自动被新的声明所继承。这使得更容易写出新的声明，而不必重复声明中相同的一部分元素，这特别适用于那些必须一致的属性。继承只作用于声明本身而不是声明的元素。

一般的规则是，如果新的声明中没有出现下列情况中的属性，原有的那些属性仍被保留。

对类和组件都有效的属性：

- public，protected
- inner，outer
- 符合 3.2.10.1 节中规则的约束类型。

只对组件有效的属性：

- flow
- discrete，parameter，constant
- input，output
- 数组维数。

[注意来自原有声明的继承性。大多数情况下，替换的或原有的属性都是没有关系的。但如果用户先将一个变量重声明为一个参数，然后再次重声明时不是参数，这就有问题了。]

[例：

```
model HeatExchanger
    replaceable parameter GeometryRecord geometry;
    replaceable input Real u[2];
end HeatExchanger;
HeatExchanger(
    /* redeclare */ replaceable /* parameter */ GeoHorizontal geometry;
    redeclare /* input */ Modelica.SIunits.Angle u /* [2] */);
    //Modelica 语义确保/*...*/中的属性自动从原有声明中加入进来
```

]

类型为“class extends B(…)”的类声明用另一个声明替换基类 B，由于该声明继承同一个基类，这时可选的类变型就作用于基类。*[由于隐含地所有声明都*

可以带变型进行继承，没必要将变型作用于新的声明。]

对于“class extends B(…)”，基类服从继承元素重声明时相同的限制，只有当新的定义可替换时，那些新的元素才是可替换的。

[例：

```
class A
  parameter Real x;
end A;
class B
  parameter Real x=3.14, y; //B 是 A 的子类型
end B;
class C
  replaceable A a(x=1);
end C;
class D
  extends C(redeclare B a(y=2));
end D;
```

这实际上产生了一个具有下列内容的类 D2：

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

继承现有 package 的例子：

```
package PowerTrain //从其他人得到的库
  replaceable package GearBoxes
    ...
  end GearBoxes;
end PowerTrain;

package MyPowerTrain
  extends PowerTrain; //使用 PowerTrain 中所有的类

  package extends Gearboxes //增加类到子库
    ...
  end Gearboxes;
end MyPowerTrain;
```

用约束类型构造的复杂类型 package 的例子：

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

  replaceable partial model BaseProperties
    ...
```

```
    end BaseProperties;

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

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

    model extends BaseProperties (T(stateSelect=StateSelect.prefer))
      //带变型从 BaseProperties 继承
      //注意 nX = 1 (!)
      …
    end BaseProperties

    function extends dynamicViscosity
      //从 dynamicViscosity 继承
      ...
    end dynamicViscosity;
```

或者

```
    redeclare function dynamicViscosity
      //用一个新的实现替换 dynamicViscosity
      ...
    end dynamicsViscosity;
  end Air;
```

]

3.2.10.1 约束类型(Constraining type)

在可替换(replaceable)声明中，可选的约束子句(constraining_clause)定义了一个约束类型。跟在约束类型名字后面的任何变型，既用于定义实际的约束类型，也自动用于当前声明以及随后的任何重声明。如果原有声明中没有约束子句(原有声明即非重声明的声明)，那么该声明的类型也被用作约束类型，声明中的变型会影响约束类型，并作用于随后的重声明。

[例：

约束类型的变型自动作用于随后的重声明：

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

```
    end ElectricalSource;

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

无约束类型的基类型变型自动作用于随后的重声明：

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

    model Circuit2
        extends Circuit(
            redeclare replaceable model NonlinearResistor
                                    = ThermoResistor(T0=300));
                //基类型变型的结果是，R 的缺省值是 100
    end Circuit2;

    model Circuit3
        extends Circuit2(
            redeclare replaceable model NonlinearResistor
                                    = Resistor(R=200));
                //T0 变型没有起作用，因为该变型没有出现在原有声明中
    end Circuit3;
```

重声明可以重新定义约束类型：

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

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

]

类或组件类型应该为约束类型的子类型。在可替换元素的重声明中，类或组件类型必须是约束类型的子类型。可替换的重声明的约束类型必须是其重声明的声明中约束类型的子类型。在可替换元素的元素变型中，变型既作用于实际类型，也作用于约束类型。

在可替换元素的元素重声明中，被替换的约束类型的变型同时合并到新的声明和新的约束类型，合并遵循外层变型覆盖内层变型的一般规则。

当一个类作为约束类型实例化时，其可替换元素的实例将使用约束类型而不是实际的缺省类型。

3.2.10.2 重声明的限制(Restriction on redeclarations)

下列附加的限制作用于重声明(在属性被继承之后，参见 3.2.10 节)：

- 只有声明为可替换的类和组件可以用新的类型进行重声明，该类型必须是原有声明约束类型的子类型，而且为允许再次进行重声明，必须使用“**redeclare replaceable**”
- 继承子句中使用的可替换类应只包含公有组件*[否则，不能保证重声明能够保持可替换缺省类的保护变量]*
- 声明为 **constant** 的元素不能被重声明
- 声明为 **parameter** 的元素只能重声明为 **parameter** 或 **constant**
- 声明为 **discrete** 的元素只能重声明为 **discrete**、**parameter** 或 **constant**
- **function** 只能重声明为 **function**
- 声明为 **flow** 的元素只能重声明为 **flow**
- 声明为非 **flow** 的元素只能重声明为非 **flow**
- 声明为 **input** 的元素只能重声明为 **input**
- 声明为 **output** 的元素只能重声明为 **output**

Modelica 不允许将保护的元素重声明为公有元素，或将 公有元素重声明为保护元素。

数组维数可以被重声明。

3.2.10.3 建 议 的 重 声 明 和 变 型 (Suggested redeclarations and modifications)

一个声明可以带有一个注解“choices”，其中包含 choice 变型，每个 choice 变型表示一个合适的重声明或元素变型。

这既是对模型用户的提示；也可用于用户界面以提示出合理的重声明，其中的字符串注释可作为选项的文字解释。注解不仅用于可替换元素，也用于不可替换元素、枚举类型和简单变量。

[例：

```
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) "...")));
```

注解也可用于不可替换的声明，例如描述枚举类型。

```
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 函数导数(Derivatives of functions)

函数声明中可带有导数注解，说明其函数导数。这会影响仿真时间和精度，可用于 Modelica 自定义函数和外部函数。导数注解表明该函数只在对输入参数作出一定限制条件下才是有效的。这些限制使用下列可选属性定义：order(只有当 order>1 时才有限制，order 缺省为 1)、noDerivative 和 zeroDerivative。所给的导数函数只能用于当这些限制满足时计算函数调用的导数。导数可以有多个限制，这种情况下所有限制必须全部满足。这些限制条件也表明，一些输入参数的导数被排除在导数调用之外(由于他们没有必要)。函数可提供多个满足受到不同限制的导数函数。

[例：

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

]

一阶导数函数的输入按下列方式构造：

首先是原函数的全部输入，接着依次为每个实型输入添加一个导数。

输出从一个空链表开始构造，然后依次为每个实型输出添加一个导数。

如果 Modelica 函数调用是第 n 阶导数(n>=1)，即该函数调用是从(n-1)阶导数求导得到的，则注解“annotation(order = n+1)=...,”声明了(n+1)阶导数，并且按下列方式构造：

使用第(n+1)阶导数添加输入参数，第(n+1)阶导数是依次第 n 阶导数依次构造而来的。

输出参数类似于第 n 阶导数的输出参数，但每个输出在导数阶数上高一阶。

[例：给定下列声明

```
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;
...

方程

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

意味着：

(..., dy(t)/dt, ...) = foo1(..., x(t), b, ..., ..., dx(t)/dt, ...);
(..., d^2 y(t)/dt^2, ...) = foo2(..., x(t), b, ..., dx(t)/dt, ..., ..., d^2(t)/dt^2, ...);

]

函数输入或输出可以是任意简单类型(实型、布尔型、整型、字符串型和枚举类型)，或者是记录(record)，如果该记录不同时包含实型和预定义的非实型。函数必须至少有一个实型输入。导数函数的输出列表不能为空。

zeroDerivative = input_var1

只有当 input_var1 独立于函数调用的微分变量时(即 input_var1 的导数为 0)，导数函数才是有效的。input_var1 的导数排除在导数函数的参数列表之外。

[假定函数 f 接受一个矩阵和一个标量。因为矩阵参数通常是参数表达式，这种参数表达式可以按下列方式定义函数(附加的 derivativ=f_general_der 是可选的，当矩阵的导数非零时可以使用)：

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

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

function f_general_der “Derivative of table lookup taking into account varying tables”
input Real x;

```
        input Real y[:, 2];
        input Real x_der;
        input Real y_der[:, 2];
        output Real z_der;
    algorithm ...
    end f_general_der;
```

]

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

导数函数只在输入参数 input_var2 按照 f(input_var1, ...)方式计算时才是有效的。input_var2 的导数排除在导数函数的参数列表之外。

[假定函数 fg 定义为复合函数 f(x, g(x))。如果第 2 个参数按如下方式定义，则可以给出对 f 进行微分时的导数：

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

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

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

如果 g 表示 fg 主要的计算效果，这种方式是有用的。]

3.2.12 受限类(Restricted classes)

关键字 class 可替换为下列关键字之一：**record**、**type**、**connector**、**model**、**block**、**package** 或 **function**。每种类定义的内容具有一定的限制。下表总结了这些限制。预定义类型在 3.6 节描述。

Record	在其定义或其组件中不允许有方程。不能用于连接。不能包含保护组件。
Type	只用于扩展预定义类型、枚举、记录或类型数组。
connector	在其定义或其组件中不允许有方程。
Model	不能用于连接。
Block	表示固定因果关系的输入—输出块。每个接口组件都是因果的，要么等于输入，要么等于输出。不能用于连接。
package	只能包含类和常量的声明。
function	具有 block 相同的限制。其它限制包括：不含方程、不含初始算法，最多有一个算法子句。可以调用的函数要求具有算法子句，或者是外部函数接口。函数中不能调用 Modelica 内置操作符 **der**、**initial**、**terminal**、**sample**、**pre**、**edge**、**change**、**reinit**、**delay**、**cardinality** 以及内部包 C**onnections** 的操作符。

3.2.13 函数类型组件(Components of function type)

函数可以被调用(参见 3.4.8 节)，或者 作为函数类型组件被实例化。也可以变型和继承一个函数类，例如为输入变量增加缺省值。

当一个函数被调用时，函数组件没有 start 属性，但是一个绑定赋值(":="表达式)作为一种表达式，在每个函数调用开始时(在执行算法段或调用外部函数之前)将函数组件初始化为该表达式。绑定赋值只用于函数组件。如果没有为非输入组件给出绑定赋值，在函数调用开始时其值是不确定的。对于 非外部函数，诊断这些是实现质量上的问题。输入参数的绑定赋值解释为缺省参数，在 3.4.8 节描述。

在函数中声明的非输入数组组件，其维数用冒号(":")指定，且没有绑定赋值，则数组维数可以按照下列特定规则进行改变。

- 在函数算法执行之前，维数大小为 õ。
- 整个数组(没有任何下标)可以用对应的任意维数大小的数组进行赋值(数组变量的维数大小是可变的)。
- 如果该组件是一个 record 组件的元素，这些规则同样适用。

[例：收集向量正值的函数

```
function collectPositive
    input Real x[:];
    output Real xpos[:];
algorithm
    for i in 1:size(x,1) loop
```

```
            if x[i]>0 then
                xpos:=cat(1,xpos,x[i:i]);
            end if;
        end for;
    end collectPositive;
```

]

对于函数的非输入数组组件，如果维数没有用冒号指定，必须用输入或常量给定。

函数组件在函数中的行为类似具有离散时间可变性。

当实例化一个函数类型组件时，其行为符合下列规则：

- 所有对于函数中组件的绑定赋值被忽略。
- 函数组件的 algorithm/external 段替换为：
  equation
  (out1,out2,...) = function call(inp1=inp1, inp2=inp2,...);
  这里函数调用的行为如前所述。
- 函数类型组件的保护组件被忽略，因为前面提到的函数调用没有为他们赋值。

[例：

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

可以用作：

```
    Real y=sin(time); //直接调用
    sin sin1; //函数类型组件
    Clock clock;
equation
    connect(clock.y, sin1.u); //连接到该对象
    //可以使用 sin1.y,
    //不能使用 sin1.x，因为 x 是保护的因而被忽略
```

这里，可以象一般函数一样调用，由于 InPort/OutPort 是连接器(connector)，也可以连接到函数类型组件。

]

3.2.14 条件组件声明(Conditional component declaration)

组件声明可以带有一个条件属性："if"表达式。

[例：

```
parameter Integer level = 1;
    Level1 component1(J=J) if level == 1 "Conditional component";
    Level component2 if level == 2, component3 if level == 3;
equation
    connect(component1..., ...) "Connection to conditional component";
    component1.u = 0; //非法
```

]

该表达式必须为布尔标量表达式，而且必须为参数表达式*[能在编译时进行求值]*。

如果该布尔表达式为 false，该组件在实例化的 DAE 中不会出现*[其变型被忽略]*，与该组件的连接也被去除。该组件的其它应用是非法的。

3.3 方程和算法(Equations and Algorithms)

3.3.1 方程与算法子句(Equations and algorithms clause)

实例化的方程或算法与未实例化的方程或算法一致。

方程或算法中的名字应能通过查找方程或算法的部分实例化父而得到。

方程等式"="不能用于算法子句，赋值运算":="不能用于方程子句。

3.3.2 If 子句(If clause)

if-和 **elseif-**子句的表达式必须是标量布尔表达式。一个 **if-**子句，零或多个 **elseif-**子句，以及一个可选的 **else-**子句一起组成一个分支列表。通过依次计算 **if-**和 **elseif-**子句中的条件，直至找到一个计算值为 true 的条件，以选取这些 **if-**、**elseif-**、**else-**子句中的一个或零个执行体。如果没有一个条件计算值为 true，则选取 **else-**子句的执行体(如果 **else-**子句存在的话，否则不选取任何执行体)。在 algorithm 段中，选取的执行体随后被执行。在 equation 段中，选取执行体中的方程被视为必须得到满足的方程。没有选取的执行体对模型计算没有影响。

在 **equation** 段中，**if-**子句中没有专门的参数表达式作为切换条件，则应具有一个 **else-**子句，而且每个分支应包含同样数目的方程*[如果这个条件违背，单一赋值规则将不成立，因为即使未知量的数目不变，在仿真期间方程数目可能改变]*。

3.3.3 For 子句(For clause)

下列结构

for IDENT **in** expression **loop**

是一个 for 子句开始部分的例子。

for 子句的表达式应为一个向量表达式。该向量表达式对每个 for 子句计算一次，并且在直接包含 for 子句的作用域内计算。在方程段中，for 子句表达式应为参数表达式。循环变量(IDENT)位于循环结构的作用域内，而且不应被赋值。循环变量具有与向量表达式的元素相同的类型。

[例：

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

循环变量可以象下例一样隐藏其它变量。但是，强烈建议为循环变量使用另外的名字。

```
    constant Integer j=4;
    Real x[j];
equation
    for j in 1:j loop  //  循环变量 j 取值 1, 2, 3, 4
        x[j] = j;            //  使用循环变量 j
    end for;
```

]

3.3.3.1 范围推导(Deduction of ranges)

没有"**in** range-expr"的迭代器"IDENT **in** range-expr"要求 IDENT 作为一个或多个下标表达式的下标出现。下标索引所在数组表达式的维数大小用于推导"range-expr"，如果下标是 Integer 子类型，"range-expr"为 1:size(其中 size 是下标所在数组表达式的大小)；如果下标是枚举类型 E=enumation(e1, ..., en)，"range-expr"为 E.e1:E.en；如果下标是 Boolean 类型，"range-expr"为 false:true。如果循环变量用于多个表达式，那么他们的范围必须是一致的。在约简表达式、数组构造表达式或 for 子句中，IDENT 可以自由地出现于非下标位置，但只作为变量 IDENT 的引用，而不用于范围推导。

[例：

```
Real x[4];
Real xsquared[:] = {x[i]*x[i] for i};
```

```
    // 等同于：{x[i]*x[i] for i in 1:size(x,1)}
    Real xsquared2[size(x,1)];
equation
    for i loop // 等同于：for i in 1:size(x,1) loop ...
        xsquared2[i] = x[i]^2;
    end for;

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

]

3.3.3.2 多个迭代器(Several iterators)

多迭代器是嵌套 for 子句或约简表达式的简化记法。对于 for 子句，将每个","换成"loop for"，并另外添加"end for"，可将其扩展成一般形式。对于约简表达式，将每个","换成") for"，并在约简表达式前面添加"function-name ("，可将其扩展为一般形式。

[例：

```
    Real x[4, 3];
equation
    for j, i in 1:2 loop
        // 循环变量 j 取值 1,2,3,4 (从使用上推导)
        // 循环变量 i 取值 1,2 (给定范围)
        x[j, i] = j + i;
    end for;
```

]

3.3.4 When 子句(When clause)

when 子句中的表达式应为离散时间的布尔型标量或向量表达式。当标量或向量表达式的任何一个元素变为 true 时，when 子句中的方程和算法语句被激活。如果 when 子句中的方程具有以下形式之一，方程段中允许使用 when 子句：

- v = expr；
- (out1, out2, out3, ...) = function_call(in1, in2, ...)；
- 操作符 **assert**()，**terminate**()，**reinit**()
- **for** 和 **if** 子句，如果 **for** 和 **if** 子句中的方程满足这些要求。

- 方程段中，when/elsewhen 不同分支中因变量必须具有相同的组件引用集合。
- 方程段中，when 子句内部的 if-then-else 子句各分支的因变量必须具有相同的组件引用集合，除非 if-then-else 子句使用不同的参数表达式作为切换条件。

when 子句不应在函数类中使用。

[例：

当 x 变为大于 2 时算法被激活：

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

当 x 变为大于 2、或 sample(0,2)变为 true、或 x 变为小于 5 时，算法被激活：

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

对于 when 在方程段中时，方程之间的顺序无关紧要。例：

```
equation
    when x > 2 then
        y3 = 2*x +y1+y2;     // 方程 y1 与 y3 的顺序不重要
        y1 = sin(x);
    end when;
    y2 = sin(y1);
```

下面的例子中，when 子句中的方程所需的限制显而易见：

```
        Real x, y;
    equation
        x + y = 5;
        when condition then
            2*x + y = 7;       // 错误：无效的 Modelica 代码
        end when;
```

当 when 子句中的方程未被激活时，哪个变量保持不变，x 或 y，是不明确的。这个例子的修正版本如下：

```
        Real x, y;
    equation
        x + y = 5;
        when condition then
            y = 7 – 2*x;       // 正确
```

```
    end when;
```

此时，当 when 子句未被激活时，变量 y 保持不变，x 使用前一事件时刻的 y 值、从第一个方程来计算。

对于算法段中的 when 子句，顺序是至关重要的。建议在 when 子句只含一个赋值，并使用多个算法段，其中包含条件一致的 when 语句。例：

```
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;
```

合并这些 when 子句将产生效率低下的代码，得到具有不同行为的不同模型，这与算法中 y1 和 y3 的赋值顺序有关。]

when 子句

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

等同于下面特殊的 if 子句，其中布尔变量 b1[N]和 b2 是必要的，因为 **edge**() 操作符只能应用于变量。

```
    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;
```

由于"**edge**(A) = A **and not pre**(A)"及其他保证条件，这个特殊的 if 子句中的算法只在事件发生瞬时进行计算。

when 子句

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

等同于下面特殊的 if 表达式

```
    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);
```

引入的布尔变量的 start 值通过取 when 条件的 start 值来定义，与前面一样 p 是一个参数变量。特殊函数 **initial**、**terminal** 和 **sample** 的 start 值为 false。

when 子句不能嵌套。

[例：

下面的 when 子句是无效的：

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

]

3.3.5 While 子句(While clause)

while 子句的表达式应为一个标量的布尔表达式。while 子句相当于编程语言中的 while 语句，形式上定义如下：

1．计算 while 子句表达式。

2．如果 while 子句表达式为 false，那么从 while 子句之后继续执行。

3. 如果 while 子句表达式为 true，那么执行 while 子句的整个循环体(除非 break 语句，被执行，见 3.3.6 节；或者 return 语句被执行，见 3.3.7 节)，然后继续执行第 1 步。

3.3.6 Break 语句(Break statement)

break 语句中断封装 break 语句的最内层 while 或 for 循环的执行，并从 while 或 for 循环之后继续执行。break 语句只能用于算法段中的 while 或 for 循环。

[例(注意这里可选择使用 return)：

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

]

3.3.7 Return 语句(Return statement)

return 语句中止当前函数调用，见 3.4.8 节。return 语句只能用在函数中的算法段。

[例子(注意这里可选择使用 break)：

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

]

3.3.8 连接(Connections)

对象之间的连接是由类的方程部分中的 **connect** 语句引入的。连接结构取两个连接器(**connector**)的引用作为参数，每个连接器引用为下列两种形式之一：

1．c1.c2. … .cn，其中，c1 是模型的一个连接器，n>=1 时，对于 i=1:(n-1)，ci+1 为 ci 的连接器元素。

2．m.c，其中 m 是模型的非连接器元素，c 为 m 的一个连接器元素。

可对任一组件选择性地使用数组下标；数组下标应为参数表达式。如果连接结构引用了连接器数组，那么数组的维数必须是匹配的，来自数组中的每一对相应的元素作为一对标量连接器进行连接。

[使用数组示例：

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

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

]

三个主要任务如下：

- 细化可扩展连接器
- 从连接(connect)语句建立连接集
- 为整个模型生成方程

定义：

可扩展连接器(Expandable connector)

如果限定符 **expandable** 出现于连接器的定义，那么该连接器的所有实例

均属于可扩展连接器。不含该限定符的连接器的实例属于非可扩展连接器。限定符 **expandable** 被继承自可扩展连接器定义的所有后续的连接器所继承。

连接集(Connection sets)

连接集是通过连接子句连接起来的变量集合。一个连接集要么只包含流(flow)变量，要么只包含非流(non-flow)变量。

内部和外部连接器(Inside and outside connections)

在一个元素实例 M 中，M 的每个连接器元素称为关于 M 的外部连接器。所有层次式包含于 M、但不是 M 的外部连接器的其他连接器，称为关于 M 的内部连接器。

[例：在 connect(a, b.c)中，"a"是一个外部连接器，"b.c"是一个内部连接器，除非"b"是一个连接器。]

3.3.8.1 可扩展连接器的细化(Elaboration of expandable connectors)

在生成连接方程之前，先细化包含可扩展连接器的连接：

当两个可扩展连接器被连接时，每个连接器均增加那些只在另一个可扩展连接器中声明的变量(这些新增的变量既不是输入也不是输出)。上述操作重复进行，直到所有连接的可扩展连接器的实例都具有匹配的变量*［即每个连接器实例都扩展成为所有连接器变量的并集］*。

下列规则应用于可扩展连接器：

- 可扩展连接器只能与其他可扩展连接器连接。
- 在细化期间引入的变量遵循缺省连接的补充规则(在 3.3.8.2 节给出)。
- 如果一个变量作为一个可扩展连接器的输入量，则在同一个扩展集合中，该变量必须在至少一个其他可扩展连接器中作为非输入量。

[例：

```
expandable connector EngineBus
end EngineBus;

block Sensor
    RealOutput speed;
end Sensor;

block Actuator
    RealInput speed;
end Actuator;
```

```
model Engine
    EngineBus bus;
    Sensor sensor;
    Actuator actuator;
equation
    connect(bus.speed, sensor.speed);       // 提供非输入量
    connect(bus.speed, actuator.speed);
end Engine;
```

]

- 可扩展连接器中的所有组件均被看作连接器实例，即使他们没有这样声明*[即有可能连接到一个 Real 变量]*。

  [例：

```
expandable connector EngineBus      // 具有预定义的信号
    import SI = Modelica.SIunits;
    SI.AngularVelocity speed;
    SI.Temperature   T;
end EngineBus;

block Sensor
    RealOutput speed;
end Sensor;

model Engine
    EngineBus bus;
    Sensor sensor;
equation
    connect(bus.speed, sensor.speed);
    // 在可扩展连接器中，连接到非连接器 speed 是合理的
end Engine;
```

  ]

- 可扩展连接器中不能包含用前缀"flow"声明的组件，但可以包含具有"flow"组件的非可扩展连接器组件。

  [例：

```
import Interfaces = Modelica.Electrical.Analog.Interfaces;
expandable connector ElectricalBus
    Interfaces.PositivePin p12, n12;    // 正确
    flow Modelica.SIunits.Current i;    // 不允许
end ElectricalBus;

model Battery
```

```
    Interfaces.PositivePin p42, n42;
    ElectricalBus bus;
equation
    connect(p42, bus.p42);   // 增加新的电气插头
    connect(n42, bus.n42);   // 增加另一个插头
end Battery;
```

]

- 连接语句中的一个连接器必须引用一个已声明的组件，而另一个未声明的组件不能有下标：
  - 可扩展连接器的实例使用一个新的组件自动进行扩展，该组件名字已被使用并且具有相应的类型。
  - 如果连接语句另一端的变量是输入或输出，那么这个新的组件也是输入或输出，以满足 3.3.8.3 节中的限制。*[如果现有的一端引用一个内部连接器(即组件的连接器)，那么新的变量将复制其因果性，即输入仍然为输入，输出仍然为输出，因为可扩展连接器必须是一个外部连接器]*。

  [例：

```
expandable connector EmptyBus
end EmptyBus;

model Controller
    EmptyBus bus1;
    EmptyBus bus2;
    RealInput speed;
equation
    connect(speed, bus1.speed);  // 正确，只有一个连接器未声明
                                 // 并且它没有下标
    connect(bus1.pressure, bus2.pressure);
                                 // 不允许，二者均未声明
    connect(speed, bus2.speed[2]);
                                 // 不允许，未声明的连接器有下标
end Controller;
```

]

经过细化之后，可扩展连接器将被当作正常的连接器实例，连接也被当作正常的连接。这种细化隐含，即使可扩展连接器不含相同的组件，它们也可以被连接。

[注意上面描述的变量引入是概念上的，并不必以任何方式影响实例层次。而且重要的是要注意这些细化规则：

1） 可扩展连接器的层次式嵌套。这意味着，在细化期间，外部和内部连接器都必须包括于层次结构的每一级。

2） 当处理一个具有 inner 作用域限定符的可扩展连接器时，细化期间所

有的 outer 实例也必须考虑。

例：

具有传感器、控制器、作动器和装置的发动机系统，通过一个总线交换信息(即通过可扩展连接器)：

```
import SI = Modelica.SIunits;
import Modelica.Blocks.Interfaces.*;
// 装置一端
model SparkPlug
    RealInput spark_advance(redeclare type = SI.Angle);
    ...
end SparkPlug;

expandable connector EngineBus
    // 无最小集
end EngineBus;

expandable connector CylinderBus
    RealInput spark_advance(redeclare type = SI.Angle); // 最小集
end CylinderBus;

model Cylinder
    CylinderBus cylinder_bus;
    SparkPlug spark_plug;
    ...
equation
    // 不需要对总线进行扩展，因为 spark_advance 在最小集中
    connect(spark_plug.spark_advance, cylinder_bus.spark_advance);
end Cylinder;

model I4 "Non array implementation"
    EngineBus ngine_bus;
    Modelica.Mechanics.Rotational.Sensors.SpeedSensor peed_sensor;
    Modelica.Thermal.Sensors.TemperatureSensor emp_sensor;
    Cylinder ylinder1;
    Cylinder ylinder2;
    Cylinder ylinder3;
    Cylinder ylinder4;
equation
    // 增加 engine_speed(作为输出)
    connect(speed_sensor.w, engine_bus.engine_speed);
    // 增加 engine_temp(作为输出)
    connect(temp_sensor.T, engine_bus.engine_temp);
    // 增加 cylinder_bus1(一个嵌套总线)
```

```
    connect(cylinder1.cylinder_bus, engine_bus.cylinder_bus1);
    // 增加 cylinder_bus2 与 cylinder_bus3
    ...
    // 增加 cylinder_bus4(一个嵌套总线)
    connect(cylinder4.cylinder_bus, engine_bus.cylinder_bus4);
    // 不允许：连接到一个带下标的未声明组件
    // connect(cylinder1.cylinder_bus, engine_bus.cylinder_bus[1]);
    //
    // 不允许：通过连接引入了多个层次(因为 cylinder_bus1 没有声明)
    // connect(temp_sensor.T, engine_bus.cylinder_bus1.eng_temp);
end I4;
```

由于有上述连接，在概念上引入了一个由所有连接器的并集所组成的连接器。engine_bus 包含下列变量声明：

```
RealOutput(redeclare type = SI.Angle)          engine_speed;
RealOutput(redeclare type = SI.Temperature)    engine_temp;
CylinderBus                                    cylinder_bus1;
CylinderBus                                    cylinder_bus2;
CylinderBus                                    cylinder_bus3;
CylinderBus                                    cylinder_bus4;
```

```
model I4_array "Array implementation"
    EngineBus engine_bus;
    Modelica.Mechanics.Rotational.Sensors.SpeedSensor speed_sensor;
    Modelica.Thermal.Sensors.TemperatureSensor temp_sensor;
    Cylinder cylinder[4];
equation
    // 增加 engine_speed(作为输出)
    connect(speed_sensor.w, engine_bus.engine_speed);
    // 增加 engine_temp(作为输出)
    connect(temp_sensor.T, engine_bus.engine_temp);
    // 带有切片，增加 cylinder_bus[4](一个总线数组)
    connect(cylinder.cylinder_bus, engine_bus.cylinder_bus);
end I4_array;
```

由于有上述连接，在概念上引入了一个由所有连接器的并集所组成的连接器。engine_bus 包含下列变量声明：

```
RealOutput(redeclare type = SI.Angle)          engine_speed;
RealOutput(redeclare type = SI.Temperature)    engine_temp;
CylinderBus                                    cylinder_bus[4];
```

]

3.3.8.2 连接方程生成(Generation of connection equations)

在生成连接方程之前，**outer** 元素被解析为实例层次中对应的 **inner** 元素(参见动态名字查找 3.1.1.3 节)。每个连接语句的参数被解析为两个连接器元素，连接在实例层次中向上移动 0 次或多次，直到两个连接器都被层次式地包含于第一个元素实例。

对于连接语句

connect(a, b);

的每一次使用，基本组件 a 和 b 形成一个连接集。如果其中任何一个组件在先前的带有匹配的内部/外部连接器的连接所形成的连接集中已经出现，则合并这些连接集形成一个连接集。复合连接器类型被分解分解为基本组件。每个连接集用于对势变量和流变量(和零)生成如下形式的方程：

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

为对流变量*[使用 flow 前缀]*生成方程，上式中用于连接器变量 zi 前面的符号，对于内部连接器为+1，对于外部连接器为-1*[如上例中的 z3]*。

对连接器的每个流变量(和零)，如果该连接器没有在任何元素实例中连接为一个内部连接器(或者，如果该变量是在可扩展连接器的扩展期间被引入的，并且该可扩展连接器没有连接为一个外部连接器)，那么隐式生成下列方程：

z = **0**;

黑体 0 表示一个具有适当维数的数组 0 或标量 0(即与 z 同维数)。

3.3.8.3 限制(Restrictions)

使用 **input** 类型前缀声明的连接器组件，至多只能在一个 **connect** 语句中作为内部连接器出现。使用 **output** 类型前缀声明的连接器组件，至多只能在一个 **connect** 语句中作为外部连接器出现。如果使用 **input** 类型前缀声明的两个组件在一个 **connect** 语句中被连接，那么一个必须是内部连接器，另一个必须是外部连接器。如果使用 **output** 类型前缀声明的两个组件在一个 **connect** 语句中被连接，那么一个必须是内部连接器，另一个必须是外部连接器。

连接器引用中的下标必须是常量表达式。

如果*[连接中的]*数组大小不匹配，那么在连接集方程生成之前，使用大小为 1 的维数从左边开始填充原有变量，直到维数匹配为止。

被连接的组件中的常量或参数生成适当的断言语句；不生成连接。

3.3.8.4 超定连接方程和虚拟连接图(Overdetermined connection equations and virtual connection graphs)

[连接器可能含有冗余变量。例如，3 维空间中两个坐标系统之间的方位可以用三个独立变量描述。然而，每个使用 3 变量的方位描述在变量定义区域内至少有一个奇异点。因此在一个连接器中不能只声明 3 个变量，改为必须使用 n(n>3) 个变量。这些变量相互之间不再独立，必须满足 n-3 个约束方程。带有约束方程的冗余变量集合的适当描述不会再有奇异点。对于由组件和带有冗余变量的连接器所形成的连接结构中有环路的模型，可能产生方程数多于未知变量数的微分代数方程系统。这些多余的方程与剩下的方程通常是相容的，即存在唯一的数学解。这样的模型不能使用当前所知的符号转换方法进行处理。为了克服这种情况，定义一些操作符使得 Modelica 翻译器能够去除多余的方程。这是通过在一定情况下以缩减的方程替换来自连接集的非流变量的等式方程而进行的。

本节处理一类特定的由于连接器带有冗余变量集而产生的超定系统。形成超定系统也有其他一些原因，如流变量显式的和零方程，这种情况不采用下面所述的方法处理。]

一个 type 或 record 声明可能具有可选的函数定义"equalityConstraint(...)"，原型如下：

```
type Type // 超定类型
    extends <base type>;
    function equalityConstraint // 非冗余等式
        input Type T1;
        input Type T2;
        output Real residue[ <n> ];
    algorithm
        residue := ...
    end equalityConstraint;
end Type;

record Record
    < declaration of record fields >
    function equalityConstraint // 非冗余等式
        input Record R1;
        input Record R2;
        output Real residue[ <n> ];
    algorithm
        residue := ...
    end equalityConstraint;
end Record;
```

equalityConstraint()函数的输出"residue"应该具有已知的大小，比方说为常数

n。该函数用非冗余的方程数 n>=0 分别表示两个 type 实例 T1 和 T2 或两个 record 实例 R1 和 R2 之间的等式关系。剩下的方程在 n 维向量"residue"中返回。描述 R1 = R2 的 n 个非冗余方程集由以下方程给出(**0** 表示一个相应大小的零向量)：

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

[如果记录 Record 的元素相互之间不是独立的，那么方程"R1=R2"含有冗余方程]。带有 equalityConstraint 函数声明的 type 类称为超定类型。带有 equalityConstraint 函数定义的 record 类称为超定记录。含有超定类型和/或超定记录类实例的连接器称为超定连接器。超定类型或记录既不能含有 flow 组件，也不能被用作 flow 组件的类型。

超定连接器中，每个超定类型或记录的实例是虚拟连接图的一个结点(**node**)。虚拟连接图用于决定何时必须采用标准方程"R1=R2"，或者方程"0 = equalityConstraint(R1, R2)"来生成 connect(...)方程。虚拟连接图的分支(**branch**)通过"connect(...)"隐式定义，或者通过"Connections.branch(...)"语句显式定义，见下表。"Connections"是一个含有内置操作符，处于全局作用域的内置包。此外，虚拟连接图的相关结点必须分别使用函数"Connections.root(...)"和"Connections.potentialRoot(...)"定义为根(**root**)或者候选根(**potential root**)。在下表中，A 和 B 为连接器实例，可能是层次结构的，例如，A 可能是"EnginePort.Frame"的缩写。

connect(A, B);	定义虚拟连接图中从连接器实例 A 的超定类型或记录实例到连接器实例 B 中对应的超定类型或记录实例的可断开分支。对应的超定类型或记录实例的类型必须相同。
Connections.branch(A.R, B.R);	定义虚拟连接图中从连接器实例 A 的超定类型或记录实例 R 到连接器实例 B 中对应的超定类型或记录实例 R 的不可断开分支。该函数可以用在 connect(...)语句允许出现的所有场合*[例如，when 子句不允许使用该函数。如果模型中有连接器 A 和 B，并且该模型中的超定记录 A.R 和 B.R 是代数耦合的，例如 B.R = f(A.R, <other unkowns>)，应使用该定义]*。
Connections.root(A.R);	定义连接器实例 A 的超定类型或记录实例 R 为虚拟连接图中的根结点。*[如果一个模型有连接器 A，超定记录 A.R 被(相容地)赋值，例如从参数表达式，该定义应被使用。]*

Connections.potentialRoot(A.R); Connections.potentialRoot (A.R, priority = p);	定义连接器实例 A 的超定类型或记录实例 R 为虚拟连接图中的候选根结点，优先级为"p"(p>=0)。如果没有给出第二个参数，优先级是 0。"p"必须是一个整型的参数表达式。在没有 Connections.root 定义的虚拟连接子图中，具有最小优先级的候选根结点被选为根结点。*[如果一个模型有连接器 A，超定记录 A.R 的微分—der(A.R)与 A.R 的约束方程一起出现，即 A.R 的非冗余子集可能用作状态变量，此时可使用该定义。]*
b = Connections.isRoot(A.R);	如果连接器实例 A 的超定类型或记录实例 R 被选作虚拟连接图的根，返回 true。

[注意，Connections.branch、Connections.root、Connections.potentialRoot 都不生成方程。它们只生成虚拟连接图中的结点和分支，用于分析目的。]

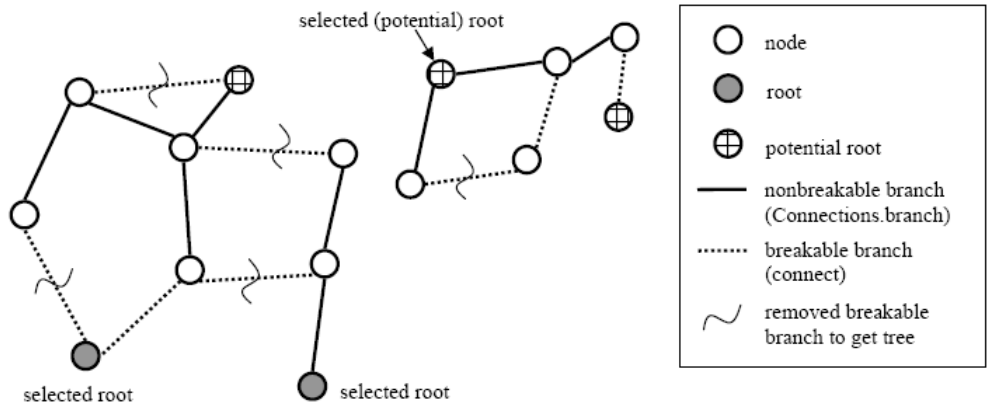
在 connect(...)方程生成之前，通过从图中移除可断开分支，将虚拟连接图转换成生成树的集合。这通过下列方式进行：

1. 通过"Connections.root(..)"语句定义的每个根结点，都是一个生成树的确定的根。
2. 虚拟连接图可由未连接在一起的子图集合所组成。该集合中的每个子图应至少有一个根结点或一个候选根结点。如果集合中的某个子图不含任何根结点，那么这个子图中具有最小优先级的候选根结点被选为该子图的根。该选择可通过在类中使用函数 Connections.isRoot(..)进行查询，见上表。
3. 如果子图中有 n 个选定的根，那么可断开分支必须被移除，这样，结果形成了一个 n 个生成树的集合，每个生成树以选定的根结点作为根。

分析完成之后，按下列方式生成连接方程：

1. 对于生成树中每个可断开分支*[即 connect(A, B)语句]*，按照 3.3.8.2 节所述生成连接方程。
2. 对于不在任何生成树中的每个可断开分支，除超定类型或记录实例 R 之外，按照 3.3.8.2 节所述生成连接方程。这里，生成方程"**0** = R.equalityConstraint(A.R, B.R) "代替"A.R = B.R"。

[虚拟连接图的例子：



]

这里有几个使用上述操作符的完整例子。

电力系统中基于Park 转换理论的超定连接器定义为：

```
type AC_Angle "Angle of source, e.g., rotor of generator"
    extends Modelica.SIunits.Angle;  // AC_Angle 是实数，单位为"rad"

    function equalityConstraint
        input AC_Angle theta1;
        input AC_Angle theta2;
        output Real residue[0] "No constraints"
    algorithm
        /* make sure that theta1 and theta2 from
          joining branches are identical */
        assert(abs(theta1 – theta2) < 1.e-10);
    end equalityConstraint;
end AC_Angle;

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

连接器中的电流和电压相对于电源的谐波高频信号来定义，该信号本质上通过电源转子的角度 theta 来描述。由于电源的基本高频信号不是微分方程的一部分，这使仿真速度会快很多。例如，当电源和其它线路的运行频率不变(=正常情况)，那么 AC_Plug.v 和 AC_Plug.i 为常量。这种情况下变步长积分器可以选择更大的时间步长。元素如3 相电感可实现为：

```
model AC_Inductor
    parameter Real X[3,3], Y[3,3]; //  组件常量
    AC_plug p;
    AC_plug n;
```

```
equation
    Connections.branch(p.theta, n.theta);   // 虚拟图中的分支
                                            // 由于 n.theta = p.theta
    n.theta = p.theta;                      // 在插脚之间传递角度 theta
    omega = der(p.theta);                   // 电源频率
    zeros(3) = p.i + n.i;
    X*der(p.i) + omega*Y*p.i = p.v – n.v;
end AC_Inductor;
```

这里，定义了电源频率，即基本变量 theta，必须使用 Connections.root(...)：

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

对虚拟连接图进行图分析，识别出连接器，这里 AC_Angle 不必在组件之间传递，以避免冗余方程。

3 维机械系统中的超定连接器可以定义为：

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

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

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

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

从标架 A 到标架 B 的固定转换定义为：

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

由于变换矩阵 frame_a.R 与 frame_b.R 代数上耦合，必须在虚拟连接图中定义一个分支。对于惯性系统，方位的初始化是相容的，因而惯性系统连接器中的方位必须定义为根：

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

3.3.9 初始化(Initialization)

在对 Modelica 模型执行任何操作之前*[例如，仿真或线性化]*，先进行初始化，对模型中出现的所有变量赋值相容的初值。在这个阶段，导数 **der**(..)和 **pre** 变量 **pre**(..)解释为未知代数变量。初始化使用目标操作*[如仿真或线性化]*中用到的所有方程和算法。初始化期间，**when** 子句中的方程当且仅当使用"**initial**()"操作符显式激活时才生效。这种情况下，**when** 子句中的方程在整个初始化阶段保持生效。*[如果 when 子句方程"v = expr;"在初始化期间没有生效，则为初始化添加方程"v = pre(v)"。这遵循 when 子句方程的映射规则。]*

对于确定所有变量的初始值所需的更多约束可按下列方式定义：

- 作为"**initial equation**"段的方程或"**initial algorithm**"段的赋值。在这些初始化段中的方程或赋值是纯代数的，描述初始时刻变量之间的约束。在

这些段中不允许使用 **when** 子句。

- 在变量声明中隐式地使用属性 start=value 和 fixed=true 隐式定义约束：
  对于所有非离散实型变量 v，如果"**start** = startExpression"和"**fixed** = **true**"，方程"v = startExpression"加入到初始化方程集。
  对于所有离散变量 vd，如果"**start** = startExpression"和"**fixed** = **true**"，方程"**pre**(vd) = startExpression"加入到初始化方程集。
  对于常量和参数，属性 fixed 缺省为 true。对于其它变量，fixed 缺省为 false。

[Modelica 翻译器首先将模型的连续方程转换成状态空间形式，至少概念上如此。这需要对方程进行微分以缩减指标，即引入附加的方程，在某些情况下还要引入附加的未知变量。整个方程系统，连同上面定义的附加约束，将产生一个代数方程系统，其中方程数和所有的变量数(包括 ***der****(..)和* ***pre****(..)变量)是相等的。通常，这是一个非线性方程系统，因而可能需要提供适当的初值(即 start 值与 fixed=****false****)，以便计算出一个数值解。*

对用户来说，可能很难计算出必须添加多少个初始方程，特别是当系统具有高阶指标时。Modelica 工具可自动增加或去除初始方程，使得结果方程系统是结构非奇异的。这些情况下，由于结果不唯一，也可能不是用户所期望的，需要提供适当的诊断机制。离散变量缺少初始值在不影响仿真结果时，可以自动设置为 start 值或缺省值，而不通知用户。例如，在 ***when*** *子句中赋值的变量，在* ***when*** *子句之外不可访问，如果* ***pre()****操作符没有显式用于这些变量，这些变量不影响仿真结果。*

例：

在稳态初始化的连续时间控制器：

```
    Real y(fixed = false); // fixed = false 是多余的
equation
    der(y) = a*y + b*u;
initial equation
    der(y) = 0;
```

初始化时具有下列解：

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

在稳态或者为状态 y 提供一个 start 值来初始化的连续时间控制器：

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

```
            der(y) = 0;
        else
            y = y0;
        end if;
```

这也可写成如下形式(这种形式不那么清晰)：

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

在稳态初始化的离散时间控制器：

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

初始化时产生下列方程：

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

其解为：

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

]

3.4 表达式(Expressions)

Modelica 方程、赋值和声明方程中包含表达式。

表达式含有基本的操作符：+、-、*、/、^、等，具有标准的优先级，如 2.2.7 节中的文法所定义。操作符的语义在 3.4.6 节定义，同时针对标量和数组参数。

也可以定义函数，并用标准的方式进行调用。标准的和命名参数的函数调用的语法在 3.4.8 节描述，向量化调用在 3.4.6.10 节中描述。内置数组函数的描述在 3.4.3 节给出，其他内置操作符的描述在 3.4.2 节给出。

3.4.1 求值(Evaluation)

Modelica 工具自行决定求解方程、重排表达式，或者不计算表达式(如果该表达式的值不影响结果，例如布尔表达式的短路求值)。If 语句和 if 表达式保证只有适当的条件为真时才计算其子句，但是，生成状态或时间事件的关系操作符在连续积分期间将保持自最近的事件以来的数值。

[例：通过 if 来保证表达式不被求值：

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

使用 noEvent 防止计算负数的平方根：

```
der(h)=if h>0 then –c*sqrt(h) else 0; // 错误
der(h)=if noEvent(h>0) then –c*sqrt(h) else 0; // 正确
```

]

3.4.1.1 纯函数(Pure function)

除了下面进一步说明的特例之外，Modelica 函数都是纯函数，即相对于 Modelica 状态(即整个仿真模型中所有 Modelica 变量的集合)没有负面效应。这意味着：

- Modelica 函数都是数学函数，即以相同的输入参数值来调用总是给出相同的结果；
- Modelica 函数相对于内部的 Modelica 仿真状态没有负面效应。特别地，函数调用的顺序和函数调用的次数不应该影响仿真状态。

[注释 1：这种特性使得能够使用 Modelica 编写陈述式的说明。对于 Modelica 编译器来说，使其可以自由地对包含函数调用的表达式进行代数处理，同时仍然保持其语义不变。]

[注释 2：Modelica 翻译器负责维护非外部纯函数的这种特性。至于外部函数，由其实现者负责维护。注意，只要不影响内部的 Modelica 仿真状态，外部函数可以具有负面效应，例如，用于提高性能或者打印跟踪信息到记录文件而缓存变量。]

- 例外情况：非纯函数只能是外部非纯函数，或者是调用某个非纯函数的 Modelica 函数。一个非纯函数(尽管有相同的输入参数值，他们可能在不同的调用中返回不同的值)可以在另一个非纯函数内部被调用，也可以在 when 子句中和在初始化期间调用。

[注释：如果这种函数调用是代数环的一部分，其语义是不确定的。

这种例外情况允许从外部硬件部分获取输入，并且/或者提供输出与这种硬件部分进行通信，例如，半实物仿真期间，在 when 子句中的事件时刻调用外部非纯函数。

以上表明，如果使用的所有外部函数都是纯函数，则所有的 Modelica 函数都为纯函数。]

3.4.2 Modelica 内置操作符(Modelica built-in operators)

Modelica 内置操作符具有与函数调用相同的语法。然而，他们的行为并不象数学函数，因为其结果不仅依赖输入参数，而且依赖仿真状态。支持以下操作符(参见 3.4.3 节的数组函数列表)：

der(expr)	expr 的时间导数。表达式 expr 必须为 Real 的子类型。表达式及其所有子表达式必须是可微的。如果 expr 是数组，那么该操作符应用到该数组中的所有元素。*[对于 Real 参数和常量，结果为标量零，或者是与变量维数相同的零数组。]*
initial()	在初始化阶段返回 **true**，否则返回 **false**。
terminal()	在分析成功结束时返回 **true**。
smooth(p, expr)	如果 p>=0，**smooth**(p, expr)返回 expr，并且说明 expr 是 p 次连续可微的，即：expr 对于出现在该表达式中所有的实型变量都是连续的，而且相对于所有的实型变量偏导数存在，并且直到 p 阶都是连续的。 smooth 中的 expr 允许的类型仅为：实型表达式，允许的表达式数组，只含有允许的表达式组件的记录。参见 3.4.2.2 节。
noEvent(expr)	按字面意义取 expr 实际基本的关系，即没有状态或时间事件被触发。参见 3.4.2.2 节和 3.5 节。
sample(start, interval)	在时间点"start + i * interval"(i = 0, 1, ...)返回 true 并且触发时间事件。在连续积分期间，该操作符总是返回 false。开始时间"start"和采样间隔"interval"要求为参数表达式，而且必须是 Real 或 Integer 的子类型。

pre(y)	返回变量 y(t)在时间点 t 的"左极限" $y(t^{pre})$。在事件时刻，$y(t^{pre})$ 为 y 在上一次事件迭代之后时刻 t 的值(参见下面注释)。若下面三个条件同时满足，则能应用 **pre** 操作符：(a) 变量 y 是简单类型的子类型；(b) y 为离散时间表达式；(c) 该操作符不能用在 **function** 类中。**pre**(y)的第一个值在初始化阶段确定。参见 3.4.2.1 节。
edge(b)	对于 Boolean 变量 b，扩展为"(b **and not pre**(b))"。适用 **pre** 操作符相同的限制(例如不能用于 function 类中)。
change(v)	扩展为"(v<>pre(v))"。适用 **pre** 操作符相同的限制。
reinit(x, expr)	在事件时刻用 expr 重新初始化状态变量 x。参数 x 要求为：(a) Real 的子类型；(b) **der** 操作符要应用于其上。expr 要求为 Integer 或 Real 表达式。reinit 操作符对同一个变量 x 只能应用一次。只能用在 when 子句内部。参见 3.4.2.3 节。
assert(condition, message)	为了使模型计算成功，条件 condition 应该为 true。其完整的描述参见 3.4.2.4 节。
terminate(message)	成功地终止当前分析。其完整描述参见 3.4.2.5 节。
abs(v)	扩展为"(**if** v>=0 **then** v **else** –v)"。参数 v 要求为 Integer 或 Real 表达式。*[注意，在 when 子句外部有状态事件被触发。]*
sign(v)	扩展为"(**if** v > 0 **then** 1 **else if** v < 0 **then** –1 **else** 0)"。参数 v 要求为 Integer 或 Real 表达式。*[注意，在 when 子句外部有状态事件被触发。]*
sqrt(v)	如果 v>=0 返回 v 的平方根，否则出现错误。参数 v 要求为 Integer 或 Real 表达式。
div(x, y)	返回代数商 x/y，分数部分舍弃(也称为向零截断)。[注意：这是 C99 为"/"定义的；在 C89 中，负数的结果由实现定义，因此必须使用标准函数 div()]。结果和参数应为 Real 或 Integer 类型，如果其中任一参数为 Real，则结果为 Real，否则为 Integer。

mod(x, y)	返回 x/y 的整型模数，即 mod(x, y)=x-floor(x/y)*y。结果和参数为 Real 或 Integer，如果其中任一参数为 Real，则结果为 Real，否则为 Integer。*[注意，在 when 子句外部，当返回值非连续变化时状态事件被触发。例如，mod(3, 1.4) = 0.2, mod(-3, 1.4) = 1.2, mod(3, -1.4) = -1.2]*
rem(x, y)	返回 x/y 的整型余数，即 div(x, y) * y + rem(x, y) = x。如果其中任一参数为 Real，则结果为 Real，否则为 Integer。*[注意，在 when 子句外部，当返回值非连续变化时状态事件被触发。例如，rem(3, 1.4) = 0.2, rem(-3, 1.4) = -0.2]*
ceil(x)	返回不小于 x 的最小整数。结果和参数为实型。*[注意，在 when 子句外部，当返回值非连续变化时状态事件被触发。]*
floor(x)	返回不大于 x 的最大整数。结果和参数为实型。*[注意，在 when 子句外部，当返回值非连续变化时状态事件被触发。]*
integer(x)	返回不大于 x 的最大整数。参数为实型，结果是整型。 *[注意，在 when 子句外部，当返回值非连续变化时状态事件被触发。]*
Integer(e)	返回枚举值 E.enumvalue 的序数，对于枚举类型 E = enumeration(e1, ..., en)，Integer(E.e1) = 1, Integer(E.en) = size(E)。

String(b, <options>) **String**(i, <options>) **String**(r, significantDigits=d, <options>) **String**(r, format = s) **String**(e, <options>)	将标量非字符串表达式转换为字符串表示。第一个参数可以为 Boolean b, Integer i, Real r 或 Enumeration e。可选的<options>如下： Integer minimumLength=0：结果字符串的最小长度。如有必要，使用空白字符来用于填充空位。 Boolean leftJustified=true：如果为 true，转换结果在字符串中左对齐；如果为 false，在字符串中右对齐。 对于 Real 表达式，输出应符合 Modelica 语法。 Integer significantDigits=6：定义结果字符串中的有效位数。*[例如："12.3456", "0.0123456", "12345600", "1.23456E-10"]*。 对应于 options 的格式字符串如下： • 对于 Real：(if leftJustified then "-" else "") + String(minimumLength) + "." + String(signficantDigits) + "g"， • 对于 Integer：(if leftJustified then "-" else "") + String(minimumLength) + "d"。 格式字符串：按照 ANSI-C 规定，格式字符串指定转换标识符(不含前面的%)，不能包含长度变型，也不能使用"*"作为宽度和/或精度。对任何浮点值，格式标识符允许为 f、e、E、g、G。对整型值，允许使用 d、i、o、x、X、u，以及 C 格式标识符(对于非整数值，如果使用整型转换字符，Modelica 工具可以圆整、截断或使用一种不同的格式)。 对于 Integer，x、X 格式(十六进制数)和 c(字符)不会导致与 Modelica 语法一致的输入。
delay(expr, delayTime, delayMax) **delay**(expr, delayTime)	当 time > time.start+delayTime 时返回 "expr(time–delayTime)"，当 time <= time.start + delayTime 时返回"expr(time.start)"。 参数 expr、delayTime 和 delayMax 要求为 Real 的子类型。delayMax 另要求为参数表达式。 下列关系应成立：0 <= delayTime <= delayMax，否则出现错误。如果参数表中未提供 delayMax，delayTime 必须为参数表达式。参见 3.4.2.7 节。
cardinality(c)	以整数的形式返回连接器实例 c(作为内部和外部连接器)在某个 connect 语句中出现的次数。参见 3.4.2.8 节。

isPresent(ident)	如果输入或输出形参 ident 作为函数调用的一个实参出现，Boolean isPresent(ident)返回 true；如果 该参数未出现，isPresent(ident)可能返回 false*[但也可能返回 true，例如对于总是计算所有结果的实现]*。isPresent()应只用于优化目的，不应影响出现在输出列表中的输出结果。该函数只能用在函数中。
semiLinear(x, positiveSlope, negativeSlope)	返回"if x>=0 then positiveSlope * x else negativeSlope * x "。结果为 Real 类型。参见 3.4.2.9 节*[特别是 x=0 的情况]*。对于非标量参数，函数按照 3.4.6.11 节所述进行向量化。

3.4.2.1 pre

在某个事件时刻，当活动模型方程求解之后，如果至少存在一个变量 v 满足"**pre**(v) <> v"，则新的事件被触发。这种情况下，模型立即被重新计算。该计算序列称为"事件迭代"。对于 **pre** 操作符中出现的所用变量 v，如果 以下条件成立："**pre**(v) == v"，那么积分重新启动。

[如果 v 和 pre(v)只出现于 when 子句，由于事件迭代中 v 不能被改变，翻译器可能为变量 v 屏蔽事件迭代。寻找最小事件迭代环是“实现质量”问题，即不是模型的所有部分都需要重新计算。

Modelica 语言允许混合代数方程系统，其中未知量为 Real、Integer、Boolean 或枚举类型。这些方程系统可以通过一种全局固定点迭代方法来求解，类似于事件迭代，在单步迭代期间固定 Boolean、Integer 和/或枚举未知量。而且，高效地求解这种系统也是一种实现质量问题，例如，将固定点迭代方法用于模型方程的一个子集。]

3.4.2.2 noEvent 与 smooth

noEvent 操作符意味着实型元素表达式按字面意义取值，而不生成 crossing 函数，参见 3.5 节。出于效率原因，应使用 smooth 操作符代替 noEvent 以避免产生事件。Modelica 工具可以不为 smooth 中的表达式生成事件。但是，smooth 不保证没有事件生成，因而有必要在 smooth 内部使用 noEvent。*[注意，如果任一变量出现非连续变化，smooth 不保证产生平滑的输出。]*

[例：

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

// noEvent 是必要的

]

3.4.2.3 重新初始化(reinit)

reinit 操作符不破坏单赋值规则，因为 **reinit**(x, expr)将先前已知的状态变量 x 变为未知变量并引入方程"x = expr"。

[如果出现了高指标系统，即状态变量之间有约束，那么一些状态变量需要重新定义为非状态变量。如果可能，非状态变量应按这样一种方式来选择，即不采用应用了 ***reinit*** *操作符的状态。如果这不可能，那么将出现错误，因为 reinit 操作符被应用于非状态变量。*

弹跳小球：

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

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

]

3.4.2.4 断言(assert)

语句

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

为断言语句，其中 condition 为布尔表达式，message 为字符串表达式。

如果断言语句的条件(condition)为真，那么消息(message)不计算，程序调用被忽略。如果 condition 计算结果为假，当前计算被中止。仿真可以继续进行另一次计算。

失败的断言先于成功的终止而生效，因此，当到达结束时间或者显式使用 terminate()，模型首先触发成功分析结束事件，但是，计算 terminate()=true 时而触发的断言，分析失败。*[目的是为了进行模型有效性测试，且如果表达式计算为假时，将失败断言报告给用户。报告失败断言的方式依赖于仿真环境。其目的是当遇到条件为假的断言时，应停止模型当前计算，Modelica 工具继续进行当前的分析(例如使用更小的步长)。]*

3.4.2.5 终止(terminate)

terminate 函数成功终止正在进行的分析，参见 3.4.2.4 节。该函数有 1 个字符串形参，表示分析成功的原因。*[目的是为了给出比时间上的固定点更加复杂的停止准则，例：*

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

]

3.4.2.6 事件触发操作符(Event triggering operators)

*[**div**、**rem**、**mod**、**ceil**、**floor**、**integer**、**abs** 和 **sign** 操作符用于 **when** 子句外部时会触发状态事件。如果不希望如此，可将 **noEvent** 函数用于其上，例如 **noEvent(abs(v))**，即|v|。*

3.4.2.7 延迟(delay)

***delay** 操作符允许一种数值上可靠的实现方式：在(内部)积分器多项式中进行插值，也允许一种更简单的实现方式：在包含表达式 expr 以前数值的缓冲中作线性插值。如果没有进一步的信息，需要存储被延迟信号完整的时间历史，因为仿真期间延迟时间可能会改变。为避免过多的存储需求并提高效率，必须通过 delayMax 给出最大允许的延迟时间。这给出了延迟信号值必须存贮的一个上限。对于采用固定步长积分器的实时仿真，该信息足以在仿真开始前为内部缓冲分配必要的存储空间。对于变步长积分器，缓冲大小在积分期间是变化的。原则上，delay 操作符能够打断代数环，但为简单起见这一点不被支持，因为最小延迟时间必须在编译期间给出，作为附加参数并且保持不变。此外，为避免在延迟缓冲中进行外插，积分器的最大步长受限于这个最小延迟时间。*

3.4.2.8 基数(cardinality)

***cardinality** 操作符允许在模型中定义连接相关的方程，例：*

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

model Resistor
  Pin p, n;
equation
  // 处理 p、n 未被连接的情况
  if cardinality(p) == 0 and cardinality(n) == 0 then
      p.v = 0; n.v = 0;
  elseif cardinality(p) == 0 then
      p.i = 0;
  elseif cardinality(n) == 0 then
      n.i = 0;
  end if;
  // Equations of resistor
  ...
end Resistor;
```

]

3.4.2.9 semiLinear

有些情况下，如果 semiLinear 函数的第一个参数(x)变成零，则带有 semiLinear 函数的方程成为欠定的，即具有无穷多的解。为了在这种情况下选择一个有意义的解，建议在翻译阶段采用以下的规则来转换方程：

规则 1：方程

```
y = semiLinear(x, sa, s1);
y = semiLinear(x, s1, s2);
y = semiLinear(x, s2, s3);
...
y = semiLinear(x, sN, sb);
...
```

替换为

```
s1 = if m_dot >= 0 then sa else sb
s2 = s1;
s3 = s2;
...
sN = sN-1;
y = semiLinear(x, sa, sb);
```

规则 2：方程

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

替换为

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

[对于符号转换，下列性质是有用的(从定义得出)：

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

等价于

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

semiLinear 函数被设计用于处理流体系统中的回流，例如

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

即，焓流速 H_dot 从质量流速 m_dot 以及依赖于流向的逆流比焓计算得到。

]

3.4.3 向量、矩阵和数组表达式的数组内置函数(Vectors, Matrices, and Arrays Built-in Functions for Array Expressions)

下列函数不能用于 Modelica，而是在下面用于定义其他操作符。

promote(A,n)	从右边开始以大小为 1 的维数填充数组 A 直到维数 n，这里要求"n>=ndims(A)"。假设 C = promote(A, n)，nA = ndims(A)，那么 ndims(C) = n；对于 1<=j<=nA，size(C, j) = size(A, j)；对于 nA+1<=j<=n，size(C, j) = 1；C[i_1, ..., i_nA, 1, ..., 1] = A[i_1, ..., i_nA]。

[函数 promote 不能用于 Modelica，因为如果 n 为变量，则返回数组的维数不能在编译时确定。下面，promote 仅用于常量 n。]

提供下列用于数组表达式的内置函数。

Modelica	*说明*
ndims(A)	返回数组表达式 A 的维数 k 的数值，k>=0。

size(A, i)	返回数组表达式 A 第 i 维的大小，这里 i>0 并且 i<=ndims(A)。
size(A)	返回长度为 ndims(A)的向量，其中包含 A 的各维大小。
scalar(A)	返回数组 A 的单个元素。 对于 1<=i<=ndims(A)，要求 size(A, i)=1。
vector(A)	如果 A 为标量，返回 1 维向量；否则，如果数组至多是一维并且大小>1，返回一个向量，其中包含数组的所有元素。
matrix(A)	如果 A 为标量或向量，返回 promote(A,2)；否则，返回一个矩阵，其中包含数组前两维的元素。对于 2<i<=ndims(A)，要求 size(A, i) = 1。
transpose(A)	转置数组 A 的前面两维。如果 A 维数少于 2，则出错。
outerProduct(v1,v2)	返回向量 v1 与 v2 的外积(=matrix(v)*transpose(matrix(v)))。
identity(n)	返回 n×n 整型单位矩阵，其对角线上的元素为 1，其他所有位置为 0。
diagonal(v)	返回一个方阵，向量 v 的元素在对角线上，其他所有元素为 0。
zeros(n1,n2,n3,...)	返回 n1 × n2 × n3 × ...的整型数组，所有元素为 0 (ni>=0)。
ones(n1,n2,n3,...)	返回 n1 × n2 × n3 × ...的整型数组，所有元素为 1 (ni>=0)。
fill(s,n1,n2,n3, ...)	返回 n1 × n2 × n3 × ...数组，其所有元素等于标量或数组表达式 s (ni>=0)。返回数组具有与 s 相同的类型。 递归定义： **fill**(s,n1,n2,n3,...)=**fill**(**fill**(s,n2,n3,...),n1);**fill**(s,n)={s,s,..., s}。
linspace(x1,x2,n)	返回具有 n 个等距元素的 Real 向量，使得 v = linspace(x1,x2,n), v[i] = x1 + (x2-x1) * (i-1)/(n-1), 1<=i<=n。要求 n>=2。参数 x1 和 x2 应为数值标量表达式。
min(A)	返回数组表达式 A 的最小元素。
min(x,y)	返回标量 x 和 y 的最小元素。
min(e(i, ..., j) **for** i **in** u, ..., j **in** v)	如同 3.4.3.1 节所述。 对于"i in u, ..., j in v"的所有组合，计算标量表达式 e(i,...,j)，返回其中的最小值。
max(A)	返回数组表达式 A 的最大元素。
max(x,y)	返回标量 x 和 y 的最大元素。

max(e(i, ..., j) **for** i **in** u, ..., j **in** v)	如同 3.4.3.1 节所述。 对于"i in u, ..., j in v"的所有组合，计算标量表达式 e(i,...,j)，返回其中的最大值。
sum(A)	返回数组表达式所有元素的标量和： A[1,...,1]+A[2,...,1]+....+A[end,...,1]+A[end,...,end]。
sum(e(i, ..., j) **for** i **in** u, ..., j **in** v)	如同 3.4.3.1 节所述。 对于"i in u, ..., j in v"的所有组合，计算表达式 e(i,...,j)的值，返回其和： e(u[1],...,v[1])+e(u[2],...,v[1]) + ... + e(u[end],...,v[1]) + ... + e(u[end], ..., v[end])。 "**sum**(e(i, ..., j) **for** i **in** u, ..., j **in** v"的类型与 e(i,...j)的类型相同。
product(A)	返回数组表达式 A 所有元素的标量积： A[1,...,1]*A[2,...,1]*....*A[end,...,1]*A[end,...,end]。
product(e(i, ..., j) **for** i **in** u, ..., j **in** v)	如同 3.4.3.1 节所述。 对于"i in u, ..., j in v"的所有组合，计算表达式 e(i,...,j)的值，返回其积： e(u[1],...,v[1])*e(u[2],...,v[1])*...*e(u[end],...,v[1])*...*e(u[end],...,v[end])。 "**product**(e(i, ..., j) **for** i **in** u, ..., j **in** v) "的类型与 e(i,...j)的类型相同。
symmetric(A)	返回一个矩阵，其对角线和对角线之上的元素等于矩阵 A 的对应元素，对角线下面的元素被设为等于 A 对角线之上的元素，即，B:=symmetric(A) -> 若 i<=j, B[i, j]:=A[i, j]; 若 i>j, B[i, j] := A[j, i]。
cross(x,y)	返回 3 维向量 x 和 y 的叉积，即， 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)	返回一个与 3 维向量关联的 3 x 3 斜对称矩阵，即，
	cross(x,y) = skew(x)*y;
	skew(x) = [0, -x[3], x[2]; x[3], 0, -x[1]; -x[2], x[1], 0];

[例：

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

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

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

]

3.4.3.1 约简表达式(Reduction expressions)

表达式

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

为约简表达式。约简表达式的 iterators 中的表达式应为向量表达式。他们对每个约简表达式计算一次，在直接包含约简表达式的作用域内进行计算。

对于 iterators

IDENT in expression2

循环变量 IDENT 在 expression1 内部的作用域中。与在 for 子句中一样，循环变量可以掩藏其它变量。结果依赖于函数名字，当前合法的函数名只能是内置操作符 array、sum、product、min 和 max。对于 array，参见 3.4.2.2 节。如果函数名是 sum、product、min 或 max，其结果与 expression1 的类型相同，构造方式如下，对循环变量的每个值计算 expression1，并计算结果元素的和、积、最小值或最大值。对于范围推导，参见 3.3.3.1 节。

函数名	**expression1 的限制**	**expression2 为空时的结果**
sum	无	zeros(...)
product	标量	1
min	标量	Modelica.Constants.inf
max	标量	-Modelica.Constants.inf

[例：

```
  sum(i for i in 1:10) // 给出 ∑_{i=1}^{10} i =1+2+...+10=55
  // 读作：计算 i 的和，i 的范围从到 10
  sum(i^2 for i in {1,3,7,6}) // 给出 ∑_{i∈{1376}} i^2 =1+9+49+36=95
  {product(j for j in 1:i) for i in 0:4} // 给出 {1,1,2,6,24}
  max(i^2 for i in {3,7,6}) // 给出 49
```

]

3.4.4 向量、矩阵和数组构造(Vector, Matrix and Array Constructors)

3.4.4.1 数组构造(Array Construction)

构造函数 array(A,B,C,...)根据其实参按照下列规则构造一个数组：

- 大小匹配：所有参数必须具有相同的大小，即，size(A)=size(B)=size(C)=...
- 所有参数必须类型等价。结果数组的数据类型是这些实参的最大扩展类型。最大扩展类型应该是等价的。Real 和 Integer 子类型可以混用，产生一个 Real 结果数组，其中 Integer 数值已被转换为 Real 数值。
- 该构造函数每应用一次就相对于实参数组的维数在结果数组的左边增加一维，即，ndims(array(A,B,C)) = ndimes(A) + 1 = ndims(B) + 1, ...
- {A, B, C, ...}是 array(A, B, C, ...)的简化记法。
- 必须至少有一个参数*[即，array()或{}没有定义]*。

[例：

```
{1,2,3}是整型3 维向量。
{ {11,12,13}, {21,22,23} }是整型 2×3 矩阵。
{{{1.0, 2.0, 3.0}}}是实型 1×1×3 数组。

Real[3] v = array(1, 2, 3.0);
type Angle = Real(unit="rad");
parameter Angle alpha = 2.0; // alpha 类型为 Real。
// array(alpha, 2, 3.0)是实型3 维向量。
Angle[3] a = {1.0, alpha, 4}; // a 的类型为 Real[3]。
```

]

3.4.4.2 带迭代器的数组构造(Array Constructors with iterators)

表达式

"{" expression **for** iterators "}"

或

array "(" expression **for** iterators ")"

为带迭代项的数组构造器。数组构造器的 iterators 中的表达式应为向量表达式。他们对每个数组构造器计算一次，在直接包含数组构造器的作用域内进行计算。

对于 iterators

IDENT **in** array_expression，

循环变量 IDENT 在 expression1 内部的作用域中。与在 for 子句中一样，循环变量可以掩藏其它变量。循环变量与 array_expression 的元素具有相同的类型。对于范围推导，参见 3.3.3.1 节。

带一个迭代器的数组构造

如果只有一个迭代器使用，其结果为向量，是通过对循环变量的每个值计算表达式并形成的结果数组来构造的。

[例：

```
array(i for i in 1:10)
// 给出向量 1:10={1,2,3,...,10}

{r for r in 1.0 : 1.5 : 5.5}
// 给出向量 1.0:1.5:5.5={1.0, 2.5, 4.0, 5.5}

{i^2 for i in {1,3,7,6}}
// 给出向量{1, 9, 49, 36}
```

]

带多个迭代器的数组构造

带多个迭代器的记法是嵌套数组构造的简化记法。通过将每个","替换为"} for"，并在数组构造器前面补充"{"，这种记法可以扩展为通常的形式。

[例：

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

]

3.4.4.3 数组连接(Array Concatenation)

函数 **cat**(k, A, B, C, ...)按照下列规则沿维数 k 连接数组 A,B,C,...：

- 数组 A,B,C,...必须具有相同数目的维数，即，ndims(A) = ndims(B) = ...
- 数组 A,B,C,...必须类型等价。结果数组的数据类型是这些实参的最大扩展类型。最大扩展类型应该是等价的。Real 和 Integer 子类型可以混用，产生一个 Real 结果数组，其中 Integer 数值已被转换为 Real 数值。
- k 必须是(这些实参数组)存在的维数，即，1 <= k <= ndims(A) = ndims(B) = ndims(C)；k 应为整数。

- 大小匹配：除了第 k 维的大小之外，数组 A,B,C,...必须具有相同的数组大小，即，对于 1 <= j <= ndims(A) 且 j <> k，size(A,j) = size(B,j)。

[例：

```
Real[2,3]   r1 = cat(1, {{1.0, 2.0, 3}}, {{4, 5, 6}});
Real[2,6]   r2 = cat(2, r1, 2*r1);
```

]

连接形式化定义如下：

假设 R = cat(k,A,B,C,...)，n = ndims(A) = ndims(B) = ndims(C) = ...，那么
size(R,k) = size(A,k) + size(B,k) + size(C,k) + ...
对于 1 <= j <= n 且 j <> k，size(R,j) = size(A,j) = size(B,j) = size(C,j) = ...，
对于 i_k <= size(A,k)，R[i_1, ..., i_k, ..., i_n] = A[i_1, ..., i_k, ..., i_n]，
对于 i_k <= size(A,k) + size(B,k)，
R[i_1, ..., i_k, ..., i_n] = B[i_1, ..., i_k - size(A,i), ..., i_n]，
....
其中，对于 1 <= j <= n，1 <= i_j <= size(R,j)。

3.4.4.4 沿第一和第二维的数组连接(Array Concatenation along First and Second Dimensions)

为了方便起见，提供一种特殊的语法用于沿第一和第二维的连接：

- *沿第一维的连接：*
  [A; B; C; ...] = cat(1, promote(A,n), promote(B,n), promote(C,n), ...)，其中，n = max(2, ndims(A), ndims(B), ndims(C), ...)。如果必要，在操作执行之前，大小为 1 的维数被加到 A,B,C 的右边，使得操作数具有相同的至少为 2 的维数。
- *沿第二维的连接：*
  [A, B, C, ...] = cat(2, promote(A,n), promote(B,n), promote(C,n), ...)，其中，n = max(2, ndims(A), ndims(B), ndims(C), ...)。如果必要，在操作执行之前，大小为 1 的维数被加到 A,B,C 的右边，特别地，使得每个操作数至少为 2 维。
- 这两种形式能够混用。[...,...]要比[...;...]优先级高，例如，[a,b;c,d]解析为[[a,b];[c,d]]。
- [A] = promote(A, max(2, ndims(A)))，即，如果 A 具有 2 或更多维数，[A] = A；如果 A 是一个标量或向量，[A]为包含 A 的元素的矩阵。
- 必须至少有一个实参(即，[]没有定义)。

[例：

```
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]是(n1+n2)×1 矩阵
[M1;M2]是(m1+m2)×n 矩阵
[M3,M4]是n×(m1+m2)矩阵
[K1;K2]是(m1+m2)×n×k 数组
[s1;s2]是2×1 矩阵
[s1,s1]是1×2 矩阵
[s1]是1×1 矩阵
[v1]是n1×1 矩阵

Real[3]    v1 = array(1, 2, 3);
Real[3]    v2 = {4, 5, 6};
Real[3,2]    m1 = [v1, v2];
Real[3,2]    m2 = [v1, [4;5;6]]; // m1 = m2
Real[2,3]    m3 = [1, 2, 3; 4, 5, 6];
Real[1,3]    m4 = [1, 2, 3];
Real[3,1]    m5 = [1; 2; 3];

]
```

3.4.4.5 向量构造(Vector Construction)

向量可以用一般的数组构造器进行构造，例如，Real[3] v = {1,2,3}。

简单表达式的范围向量操作符或冒号操作符可以用于代替或组合这种一般的数组构造器，来构造 Real、Integer、Boolean 或枚举类型的向量。冒号操作符的语义为：

- 如果 j 和 k 是 Integer 类型，j : k 为 Integer 类型向量{j, j+1, ..., k}。
- 如果 j 和/或 k 是 Real 类型，j : k 是 Real 类型向量{j, j+1.0, ... n}，其中 n = floor(k-j)。
- 如果 j > k，j : k 为 Real、Integer、Boolean 或枚举类型的具有 0 个元素的向量。
- 如果 j、d 和 k 都为 Integer 类型，j : d : k 为 Integer 类型向量{j, j+d, ..., j+n*d}，其中 n = (k – j)/d。
- 如果 j、d 或 k 为 Real 类型，j : d : k 为 Real 类型向量{j, j+d, ..., j+n*d}，其中 n = floor((k – j)/d)。
- 如果 d > 0 且 j > k，或者如果 d < 0 且 j < k，j : d : k 为 Real 或 Integer 类型的具有 0 个元素的向量。
- false : true 为 Boolean 类型向量{false, true}。

- 如果 j 为 Real、Integer、Boolean 或枚举类型，j:j 为{j}。
- E.ei : E.ej 为枚举类型向量{ E.ei, ... E.ej }，这里 E.ej > E.ei，并且 ei 和 ej 属于某个枚举类型 E=enumeration(...ei,...ej,...)。

[例：

```
Real v1[5] = 2.7 : 6.8;
Real v2[5] = {2.7, 3.7, 4.7, 5.7, 6.7}; // 等同于 v1
Boolean b1[2] = false:true;
Colors = enumeration(red,blue,green);
Colors ec[3] = Colors.red : Colors.green;
```

]

3.4.5 数组访问操作符(Array access operator)

向量、矩阵或数组变量的元素使用"[]"访问。冒号用于表示某一维所有下标。向量表达式用于挑拣出向量、矩阵和数组中选定的行、列和元素。该表达式维数通过标量下标实参的数目推导。表达式 end 只能出现于数组下标中，如果用于数组表达式 A 的第 i 个下标，假设 A 的下标为 Integer 子类型，那么它等价于 size(A,i)。如果用于嵌套的数组下标中，它指向最近的嵌套数组。

可以使用数组访问操作符对算法段中的一个/多个数组元素赋值。如果下标是数组，赋值按下标数组给定的顺序进行。对于数组或数组元素的赋值，整个右边与左边的下标在任何元素被赋新值之前计算。

[例：

- a[:, j]是 a 的第 j 列向量，
- a[j : k]是{[a[j], a[j+1], ... , a[k]}
- a[:,j : k]是[a[:,j], a[:,j+1], ... , a[:,k]],
- v[2:2:8] = v[{2,4,6,8}] .
- v[{j,k}]:={2,3}; // 等同于 v[j]:=2; v[k]:=3;
- v[{1,1}]:={2,3}; // 等同于 v[1]:=3;
- A[**end**-1,**end**]为 A[size(A,1)-1,size(A,2)]
- A[v[**end**],**end**]为 A[v[size(v,1)],size(A,2)] // 因为第一个 end 引用 v 的 end。
- 如果 x 是向量，x[1]是标量，但是切片 x[1:5]是向量(矢量值或冒号下标表达式导致一个向量被返回)。

]

[例，假设 x[n, m]、v[k]、z[i, j, p] 已声明：

表达式	#维数	结果类型
x[1, 1]	0	标量
x[:, 1]	1	n 维向量

x[1, :]	1	m 维向量
v[1:p]	1	p 维向量
x[1:p, :]	2	p×m 矩阵
x[1:1, :]	2	1×m"行"矩阵
x[{1, 3, 5}, :]	2	3×m 矩阵
x[: , v]	2	n×k 矩阵
z[: , 3, :]	2	i×p 矩阵
x[scalar([1]), :]	1	m 维向量
x[vector([1]), :]	2	1×m"行"矩阵

]

3.4.6 标量、向量、矩阵和数组操作函数(Scalar, Vector, matrix, and array operator functions)

定义于标量、向量和矩阵上的数学操作属于线性代数范畴。

在所有需要 Real 子类型表达式的上下文中，Integer 子类型的表达式也可以使用；Integer 表达式被自动转换为 Real。

术语数值类型在下面用于指 Real 或 Integer 类型的子类型。

3.4.6.1 类型类的等式与赋值(Equality and Assignment of type classes)

标量、向量、矩阵和数组的等式"a=b"与赋值"a:=b"是基于元素定义的，并且要求两个对象具有相同的维数和对应的维数大小。操作数要求类型等价。对于简单类型和满足记录要求的所有类型这是合法的，后一种情况下适用于记录的每个组件元素。

a 的类型	b 的类型	a = b 的结果	操作 (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 数值类型类的加减与字符串连接(Addition and subtraction of numeric type classes and concatenation of strings)

数值标量、向量、矩阵和数组的加"a+b"与减"a-b"是基于元素定义的，并要求 size(a) = size(b)，a 和 b 均为数值类型类。字符串标量、向量、矩阵和数组的加"a+b"定义为从 a 到 b 的对应元素逐个字符串连接，并要求 size(a) = size(b)。

a 的类型	b 的类型	a+/-b 的结果	操作 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 数值类型类的标量乘法(Scalar Multiplication of numeric type classes)

数值标量 s 与数值标量、向量、矩阵或数组 a 的标量乘法"s*a"或"a*s"是基于元素定义的：

s 的类型	a 的类型	s*a 和 a*s 的类型	操作 c:=s*a 或 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 数值类型类的矩阵乘法(Matrix Multiplication of numeric type classes)

数值向量和矩阵的乘法"a*b"只针对下列组合定义：

a 的类型	b 的类型	a*b 的类型	操作 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

[例：

```
Real A[3,3], x[3], b[3], v[3];
A*x = b;
x*A = b; // 等同于 transpose([x])*A*b
[v]*transpose([v]) // 外积
v*A*v // 标量
tranpose([v])*A*v // 只有一个元素的向量
```

]

3.4.6.5 数值类型类的标量除法(Scalar Division of numeric type classes)

数值标量、向量、矩阵或数组 a 与数值标量 s 的除法"a/s"是基于元素定义的。结果总是 Real 类型。为了得到带有截断的整数除法，可使用函数 **div**。

a 的类型	b 的类型	a/s 的结果	操作 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 数值类型类标量的求幂(Exponentiation of Scalars of numeric type classes)

如果"a"和"b"都是数值类型类的标量，求幂"a^b"定义为 C 语言中的 pow()。

3.4.6.7 数值类型类方阵的标量求幂(Scalar Exponentiation of Square Matrices of numeric type classes)

如果"a"是一个数值方阵，"s"是 Integer 子类型的标量，并且 s>=0，求幂"a^s"被定义。求幂通过反复相乘进行(例如，a^3 = a*a*a; a^0 = identity(size(a,1)); assert(size(a,1)==size(a,2), "矩阵必须是方阵"); a^1 = a)。

[非整型指数是不允许的，因为这需要计算"a"的特征值和特征向量，不再是一种元素操作。]

3.4.6.8 切分操作(Slice operation)

如果 a 为包含标量组件的数组，m 是其中的一个组件，表达式 a.m 解释为一种切分操作，返回组件数组{a[1].m, ...}。

如果 m 也是一个数组组件，切分操作只有当 size(a[1].m)=size(a[2].m)=...时才有效。

3.4.6.9 关系操作符(Relation operators)

关系操作符<、<=、>、>=、==、<>只针对简单类型的标量操作数来定义。结果是 Boolean 类型，根据关系是否满足分别取 **true** 或 **false**。

对于字符串类型的操作数，对每个关系操作符 op，"str1 op str2"按照 C 函数 strcmp 定义为"strcmp(str1, str2) op 0"。

对于 Boolean 类型的操作数，false<true。

对于枚举类型的操作数，运算规则由枚举常量声明的顺序给出。

除非用于函数中，形如 v1 == v2 或 v1 <> v2 的关系中，v1 或 v2 不能为 Real 的子类型。

[该规则的理由是，带有 Real 实参的关系被转换为状态事件(参见下面的 3.5 节，事件)，这种转换对于==和<>关系操作符变得不必要的复杂(例如，需要两个 crossing 函数代替一个 crossing 函数，甚至在事件时刻需要采用 epsilon 策略)。而且，Real 变量的相等检验在寄存器中数值长度不同于内存中数值长度的计算机上是有问题的。]

形如"v1 rel_op v2"的关系被称为基本关系，其中 v1 和 v2 为变量，rel_op 为关系操作符。如果 v1 或 v2 之一或者两者都为 Real 子类型，该关系被称为 Real 元素关系。

3.4.6.10 布尔操作符(Boolean operators)

操作符"and"和"or"包括多个 Boolean 类型表达式，可以是标量或维数匹配的数组。操作符"not"包括一个 Boolean 类型的表达式，可以是标量或数组。结果为按位逻辑操作。对于"and"和"or"的短路计算，参见 3.4.1 节。

3.4.6.11 函数的向量化调用(Vectorized call of functions)

只有一个标量返回值的函数可以按元素的方式应用于数组，例如，如果 A 是一个 Real 向量，那么 sin(A)是一个向量，其每个元素是应用函数 sin 到 A 中对应元素的结果。

考虑表达式 f(arg1, arg2, ..., argn)，函数 f 对于实参 arg1, arg2, ..., argn 的应用定义如下。

对于每个传入的实参，其参数类型按照函数对应形参的类型进行检查。

1. 如果类型匹配，不做任何事情；
2. 如果类型不匹配，但可应用类型转换，那么应用转换。转到步骤 1 继续；
3. 如果类型不匹配，且无类型转换可用，那么检查传入实参的类型，看其是否为一个形参类型的 n 维数组。如果不是，函数调用无效。如果是，我们称此为一个 foreach 参数。
4. 对于所有的 foreach 参数，他们的维数和大小必须匹配。如果他们不匹配，函数调用无效。如果不存在 foreach 参数，函数按正常风格调用，结果具有函数定义所规定的类型。
5. 函数调用表达式的结果是一个与 foreach 参数维数相同的 n 维数组。其中每个元素 ei, ..., j 为将 f 应用到从原始参数按下列方式构造的参数的结果：
   - 如果参数不是 foreach 参数，按其原来的形式使用。
   - 如果参数是 foreach 参数，使用下标为[i, ..., j]的元素。

如果不止一个参数为数组，那么所有参数必须具有相同的大小，并且并行地被遍历。

[例：

```
sin({a, b, c})          = {sin(a), sin(b), sin(c)}      // 参数是向量
sin([a,b,c])            = [sin(a),sin(b),sin(c)]        // 参数可以是矩阵
atan({a,b,c},{d,e,f}) = {atan(a,d), atan(b,e), atan(c,f)}
```

即使函数被声明为取一个数组作为其参数之一，上述规则起作用。如果 pval 定义为取一个 Real 向量为参数的函数，返回一个 Real，那么该函数可以使用一个二维数组(向量的向量)作为其实参。这种情况下，结果类型将为 Real 向量。

```
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])将 1 加到第二个参数的每一个元素，得到结果[2, 3, 4]。但是，直接写 1+[2, 3, 4]是不合法的，因为内置操作符的规则有更多的限制。]

3.4.6.12 空数组(Empty arrays)

数组可以具有大小为 0 的维数，例如

```
Real x[0];                      // 空向量
Real A[0, 3], B[5, 0], C[0, 0]; // 空矩阵
```

- 空矩阵可以使用 fill 函数构造。例如：

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

- 不能访问空矩阵的元素，例如，如果"v=[]"，v[j, k]是错误的，因为断言"下标必须大于 1"失败。
- 即使维数为零，类似"+"、"-"运算的大小要求也必须满足。例如：

```
Real[3,0] A, B;
Real[0,0] C;
A + B // 正确，结果是空矩阵
A + C // 错误，维数不一致
```

- 如果结果矩阵没有大小为零的维数，两个空矩阵相乘产生零矩阵，例如：

```
Real[0,m]*Real[m,n] = Real[0,n] (空矩阵)
Real[m,n]*Real[n,0] = Real[m,0] (空矩阵)
Real[m,0]*Real[0,n] = zeros(m,n) (非空矩阵，有零元素).
```

[例：

```
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
```

*假设 n=0, p>0, q>0，结果为"y = D*u"。*

]

3.4.7 If 表达式(If-expression)

表达式

if expression1 **then** expression2 **else** expression3

是 if 表达式的一个例子。expression1 必须为布尔表达式并且首先被计算。如果 expression1 为真，expression2 被计算并作为 if 表达式的值，否则 expression3 被计算并作为 if 表达式的值。两个表达式 expression2 和 expression3 必须类型兼容，且给出了 if 表达式的类型。带 elseif 的 if 表达式通过用 else if 替换 elseif 来定义。*[注意：elseif 是为了 if 子句对称而增加的。]*关于短路计算参见 3.4.1 节。

[例：

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

]

3.4.8 函数(Functions)

函数类和记录构造函数可以按本节所描述的方式来调用。也可以声明函数类型的组件，参见 3.2.13 节。

3.4.8.1 函数的输入形参(Formal input parameters of functions)

函数调用带有位置参数和命名参数，位置参数可选，后面跟 0 个、1 个或多个命名参数，参见 2.2.7 节，例如

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

函数调用的解释如下：首先，为所有输入形参创建一个未填充的槽的列表。如果有 N 个位置参数，他们被放入前面 N 个槽中，参数顺序由函数定义中的组件声明的顺序给出。然后，对于每个命名参数"identifier = expression"，identifier 用于确定对应的槽。该槽应该没有被填充*[否则出现错误]*，将实参的值放入该槽中填充它。当所有参数都被处理之后，那些仍未填充的槽由函数定义中对应的缺省值来填充。这是应该没有剩下未填充的槽*[否则出现错误]*，已填充的槽的列表用作函数调用的实参列表。

每个实参的类型必须与对应形参的类型一致，除非可以应用标准类型强制转换使得类型一致。(参见 3.4.6.10 节，关于将标量函数应用到数组。)

[例：

假设如下定义函数 RealToString，将 Real 数值转换为 String：

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

那么下列函数调用是等价的：

```
RealToString(2.0);
RealToString(2.0, 6, 0);
RealToString(2.0, 6);
RealToString(2.0, precision=6);
RealToString(2.0, length=0);
RealToString(2.0, 6, precision=6); // 错误，槽被使用了两次
```

]

3.4.8.2 函数的输出形参(Formal output parameters of functions)

函数可以具有多个输出组件，对应于多个返回值。调用 一个函数返回多个结果的唯一方式是，将函数调用放在方程或赋值的右边。这种情况下，方程或赋值的左边应该包含一个由圆括号包围的组件引用列表：

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

组件引用按照它们在列表中的位置与输出组件关联，这样，输出组件 i 被设置等于或赋值给列表中的组件引用 i，其中，输出组件的顺序由函数定义中组件声明的顺序给定。列表中每个组件引用的类型必须与对应输出组件的类型一致。

如果只提供唯一一个返回结果，函数调用可以用作表达式，其值与类型由第一个输出组件的值与类型给定。

可以省略左边的组件引用以及/或者截断左边的列表，以便丢弃函数调用的输出。*[操作符 isPresent(out2)可以在函数内部使用，以确定输出形参 out2 是否需要被计算，参见 3.4.2 节。]*

[例：

函数"eigen"用于计算特征值与可选的特征向量，可按下列方式调用：

```
        ev  = eigen(A);              // 用于表达式中计算特征值
        x   = isStable(eigen(A));    //
   (ev, vr) = eigen(A)               // 计算特征向量
(ev, vr, vl) = eigen(A)              // 有左边的特征向量
(ev,    ,vl) = eigen(A)              // 无右边的特征向量
```

该函数可定义如下：

```
function eigen "calculate eigenvalues and optionally eigenvectors"
  parameter  Integer  n = size(A,1);
  input      Real     A[:, size(A,1)];
  output     Real     eigenValues[n,2];
  output     Real     rightEigenVectors[n,n];
  output     Real     leftEigenVectors [n,n];
algorithm
  // compute eigenvalues
  if isPresent(rightEigenVectors) then
      // compute right eigenvectors
  end if;
  if isPresent(leftEigenVectors) then
      // compute left eigenvectors
  end if;
end eigen;
```

]

以圆括号中表达式列表的形式出现的表达式，其唯一允许的使用方式是将该表达式用于方程或赋值的左边，同时右边是一个函数调用。

[例，以下都是非法的：

```
(x+1, 3.0, z/y) = f(1.0, 2.0);      // 不是组件引用列表。
(x, y, z) + (u, v, w)               // 没有合适的方程与/或赋值语句的 LHS。
```

]

3.4.8.3 记录构造(Record constructors)

只要一个记录(record)被定义，与该记录类的名字相同并且在相同的作用域的记录构造函数也按照下列规则被隐式定义：

- 记录声明被部分实例化，包括继承、变型、重声明，并将所有引用记录作用域外部声明的名字扩展为其完全限定性名字*[以便消除记录构造函数中由于平坦化继承树而导致的 import 语句冲突]*。
- 部分实例化的记录声明中，所有的记录元素[即组件和局部类定义]都被用作记录构造函数中的声明，但下列情况除外：(1) 不允许变型的组件声明*[例如"**constant** Real c=1"或"**final parameter** Real"]*被作为记录构造函数中的保护组件；(2) 记录中其余组件的前缀(**constant**, **parameter**, **final**, **discrete**, **input**, **output**, ...)被删除；(3) 将前缀"input"加到记录构造函数中的公有组件上。
- 记录的实例及其变型一起被声明为输出参数*[使用未出现在记录中的某个名字]*。变型中所有的输入参数用于设置对应的记录变量。

[这允许构造一个记录实例，带有可选的变型，可用在函数调用允许的任何

地方。例：

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

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

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

下面的例子中，建立了一个方便的组件数据表库：

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

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

  record MotorI123    = MotorData( // data of a specific motor
      inertia         = 0.001,
      nominalTorque   = 10,
      maxTorque       = 20,
      maxSpeed        = 3600) "Data sheet of motor I123";
  record MotorI145    = MotorData( // data of another specific motor
      inertia         = 0.0015,
      nominalTorque   = 15,
      maxTorque       = 22,
      maxSpeed        = 3600) "Data sheet of motor I145";
end Motors

model Robot
```

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

下面的例子说明了隐式记录构造函数创建可以出现的大多数情况。

使用下面的记录定义：

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

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

隐式定义了下列记录构造函数：

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

    function Record2
        input   Real    r0 = 0;
        input   Real    c2;
        input   Integer n1 := 5;
        input   Integer n2;
```

```
    input   Real     r1 "comment"; // the comment also copied from record
    input   Real     r2 := Modelica.Math.sin(c1);
    input   Real     r4;
    input   Real     r5 = 5.0;
    input   Real     r6[n1];
    input   Real     r7[n2];
    output  Record2  'result'(r0=r0,c2=c2,n1=n1,n2=n2,r1=r1,r2=r2,
                              r4=r4,r5=r5,r6=r6,r7=r7);
  protected
    constant        Real c1 = 2.0; // referenced from r2
    final parameter Real r3 = Modelica.Math.cos(r2);
  end Record2;
end Demo;
```

并且可以按下列方式使用：

```
Demo.Record2 r1 = Demo.Record2(r0=1, c2=2, n1=2, n2=3, r1=1, r2=2,
                               r4=5, r5=5, r6={1,2}, r7={1,2,3});
Demo.Record2 r2 = Demo.Record2(1,2,2,3,1,2,5,5,{1,2},{1,2,3});
parameter Demo.Record2 r3 = Demo.Record2(c2=2, n2=1, r1=1, r4=4,
                                         r6=1:5, r7={1});
```

上面的例子只用来显示使用前缀的不同形式，但不是非常有意义，因为过于简单，只使用了一个直接的变型。

]

3.4.8.4 类型转换构造器(Type conversion constructors)

类型转换函数 Integer(E.enumvalue)返回枚举值 E.enumvalue 的序数，对于枚举类型 E = enumeration(e1, ..., en)，Integer(E.e1) = 1，Integer(E.en) = size(E)。

String(E.enumvalue)给出枚举值的字符串表示。

[例：String(E.Small)得到"Small"。]

3.4.9 表达式的可变性(Variability of Expressions)

常量(**Constant**)表达式：

- Real、Integer、Boolean、String 和枚举常量。
- 声明为 **constant** 的变量。
- 除了特定的内置操作符 **initial**、**terminal**、**der**、**edge**、**change**、**sample**、**pre** 和 **analysisType**，以常量子表达式作为实参的函数或操作符(函数中没有形参定义)为常量表达式。

参数(**Parameter**)表达式：

- 常量表达式。
- 声明为 **parameter** 的变量。
- 除了特定的内置操作符 **initial**、**terminal**、**der**、**edge**、**change**、**sample** 和 **pre**，带有参数子表达式的函数或操作符为参数表达式。

离散时间(**Discrete-time**)表达式：

- 参数表达式。
- 离散时间变量，即 Integer、Boolean、String 变量和枚举变量，以及在 when 子句中赋值的 Real 变量。
- 所有输入实参为离散时间表达式的函数调用。
- 所有子表达式为离散时间表达式的表达式。
- **when** 子句中的表达式。
- 除非在 **noEvent** 内部，有序关系(>、<、>=、<=)和函数 **ceil**、**floor**、**div**、**mod**、**rem**、**abs**、**sign***[是离散时间表达式]*。如果有连续时间子表达式，这些将产生事件。*[换句话说，**noEvents()**内部的关系为连续时间表达式，例如**noEvent(**x>1)。]*
- 函数 **pre**、**edge** 和 **change** 产生离散时间表达式。
- 函数中的表达式其行为就象离散时间表达式。

声明为常量(**constant**)的组件，如果用在模型(model)中，应具有一个带有常量表达式的关联的声明方程。常量的值在给定之后不能被改变。没有关联的声明方程的常量可以通过变型给定一个值。

对于赋值 v := expr 或声明方程 v = expr，v 必须声明为至少其可变性与 expr 一样。

- 参数组件和基本类型属性*[例如 start]*的声明方程要求为参数表达式；
- 如果 v 为离散时间组件，那么 expr 要求为离散时间表达式。

对于方程 expr1 = expr2，如果两个表达式都不是基本类型 Real，那么二者都必须离散时间表达式。对于记录方程，在应用这种检查之前，方程被分解到基本类型。*[这个限制保证了**noEvent()**操作符不能应用于 when 子句之外的 Boolean、Integer、String 或枚举方程，因为这样其中一个表达式不是离散时间的。]*

在由连续时间的开关表达式控制的、并且不在 when 子句里面的 if 表达式、if 子句或 for 子句内部，对离散变量的赋值、离散时间表达式之间的方程，或者那些应该生成事件的 Real 元素关系/函数都是非法的。*[这个限制是有必要的，以保证不存在离散变量的连续时间方程，并且确保 crossing 函数不会在事件之间激活。]*

[例：

```
model Constants
  parameter   Real p1 = 1;
  constant    Real c1 = p1 + 2;     // 错误，非常量表达式
  parameter   Real p2 = p1 + 2;     // 正确
end Constants;

model Test
  Constants c1(p1=3);       // 正确
  Constants c2(p2=7);       // 正确，声明方程可以变型
  Boolean b;
  Real    x;
equation
  b = noEvent(x > 1)   // 错误，因为 b 是离散时间的
                       // 但是 noEvent(x > 1)是连续时间表达式
end Test;
```

]

3.5 事件与同步(Events and Synchronization)

只要 Real 元素关系，例如"x > 2"，改变其值，积分就被中止，并产生一个事件。关系的值只能在事件时刻改变*[换句话说，Real 元素关系引起状态或时间事件]*。在事件时刻模型被处理之前，触发事件的关系在按字面意义求值时改变其值*[换句话说，需要一种求根机制，用于确定关系改变其值的小时间区间；事件发生在这个区间的右侧]*。when 子句中的关系总是按字面意义取值。在连续积分期间，Real 元素关系具有自上一个事件时刻以来的常量关系值。

[例：

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

在连续积分期间，总是计算相同的 if 分支。只要"u-uMax"或"u-uMin"穿越零，积分就被中止。在事件时刻，选择正确的 if-分支，并重新开始积分。

n 阶(n>=1)数值积分方法要求连续的模型方程直到 n 阶都是可微的。如果 Real 元素关系不按字面意义处理而是如上定义，这个要求能够满足，因为非连续改变只能出现在事件时刻，在连续积分期间不再出现。]

[下列特殊的关系属于一种实现质量的问题，

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

在"time = discrete expression"时触发一个时间事件，即，事件时刻是预先已知的，不需要迭代来寻找精确的事件时刻。]

如果关系或含有关系的表达式是 **noEvent**(...)函数的实参，在连续积分期间，关系也按字面意义取值。**smooth**(p, x)操作符也允许用作实参的关系按字面意义

取值。这种 noEvent 特征被传播到 noEvent 函数作用域内的所有子关系。对于 smooth，不允许按字面计算的特权传播到所有子关系，但 smooth 性质本身不会被传播。

[例：

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

这个例子中 x = y = z，但是 Modelica 工具可能为 z 生成事件。这里 if-表达式按字面意义取值而不引起状态事件。

***smooth** 函数是有用的，例如，如果建模者能够保证所用的 if 子句满足积分器最低的连续性要求。这种情况下仿真速度会提高，因为积分期间没有发生状态事件迭代。**noEvent** 函数用于保证没有"越界(outside domain)"错误，例如，y = **if noEvent**(x >= 0) **then** sqrt(x) **else** 0。]*

when 子句中的所有方程和赋值语句，以及函数(function)类中的所有赋值语句都隐式地使用 **noEvent** 函数进行处理，即，这些操作符作用域内的关系绝对不会生成状态或时间事件。*[在 when 子句中使用状态事件是不必要的，因为在连续积分期间时 when 子句不被计算。]*

[例：

```
Limit1 = noEvent(x1 > 1);
// 错误，因为 Limit1 是离散时间变量
when noEvent(x1>1) or x2>10 then
  // 错误，when 条件不是离散时间表达式
  Close = true;
end when;
```

]

Modelica 基于同步数据流原理(synchronous data flow principle)，其定义如下：

1. 所有变量保持其实际的值，直到这些值被显式改变。在连续积分期间的任何时刻以及事件时刻，变量值是可以访问的。
2. 在每个时间点，包括连续积分期间和事件时刻，生效的方程表示变量之间必须同时满足的关系(如果对应的 if-分支、when-子句或出现方程的块(block)当前不是活动的，那么对应的方程也不是活动的)。
3. 在事件时刻的计算和通信不占用时间。*[如果计算或通信时间必须仿真，那么这种性质必须被显式建模。]*
4. 方程的总数恒等于未知变量的总数(=单赋值规则)。

[这些规则保证变量总是被唯一的方程集所定义。例如，不可能一个变量被两个方程定义，这将导致冲突或不确定的行为。而且，模型的连续部分与离散部

分总是自动"同步"。

例：

```
equation // 非法的例子
  when condition1 then
      close = true;
  end when;
  when condition2 then
      close = false;
  end when;
```

这不是一个有效的模型，由于有两个方程针对单一未知变量 close，规则 4 被违背。如果这是有效的模型，那么由于两个方程之间的优先级没有指定，在同一时刻点当两个条件都变为真时将出现冲突。为使其有效，模型必须被改为：

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

这时，即使在同一时刻点两个条件都变为真(condition1 比 condition2 具有更高的优先级)，模型也是定义正确的。]

不保证在同一时刻点出现两个不同的事件。

[因而，事件的同步必须在模型中显式进行编程，例如通过计数器。

例：

```
  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 // 快速采样
      ...
  end when;

algorithm
```

```
  when slowSample then //  缓慢采样(慢 5 倍)
      ...
  end when;
```

在 fastSample when 子句每出现五次时，slowSample when 子句才被计算。]

[单赋值规则和显式对事件同步进行编程的要求已经在编译时允许一定程度的模型验证。]

3.6 预定义类型(Predefined types)

(尽管)预定义变量类型和枚举类型是预定义的，其属性使用 Modelica 语法描述如下。对这些类型的任何一个进行重声明都是错误的，(类型)名字是保留字，使用这些名字声明一个元素是非法的。而且，从预定义类型的继承和其他组件不能进行组合。下列定义使用 RealType、IntegerType、BooleanType、StringType、EnumType 作为对应到机器表示的助记符。*[因此，声明象 Real 的子类型的唯一方式是使用继承机制。]*

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

type Integer
  IntegerType value;                           // Accessed without dot-notation
  parameter StringType    quantity = "";
  parameter IntegerType   min=-Inf, max=+Inf;
  parameter IntegerType   start  = 0;          // Initial value
  parameter BooleanType   fixed = true,        // default for parameter/constant;
                                = false;       // default for other variables
equation
  assert(value >= min and value <= max, "Variable value out of limit");
end Integer;
```

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

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

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

对于每个枚举

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

在概念上定义了一个新的简单类型，如下：

```
type E
  EnumType value;                             // Accessed without dot-notation
  parameter  StringType    quantity = "";
  parameter  EnumType      min=e1, max=en;
  parameter  EnumType      start = e1;        // Initial value
  parameter  BooleanType fixed = true,        // default for parameter/constant;
                               = false;       // default for other variables
  constant   EnumType      e1=...;
  ...
  constant   EnumType      en=...;
equation
  assert(value >= min and value <= max, "Variable value out of limit");
end E;
```

属性"start"和"fixed"为 analysisType = "static"(静态分析)定义了变量的初始条件。"fixed = false"表示一个初值估计，即，变量值可能被静态分析器改变。"fixed=true"表示必需的初值。在其它分析(例如"动态 dynamic")进行之前，静态分析必须首先进行。结果得到所有模型变量的相容值的集合，用作进一步分析的初始值。

属性"nominal"给出了变量的名义值。尽管标准没有定义缺省的名义值，用户也不需要设置该属性。*[名义值可以被分析工具用于确定合适的容差或 epsilon，也可以用于缩放比例。例如，积分器的绝对容差可被计算为"absTol = abs(nominal) * relTol/100"。缺省的名义值没有提供，以防止象"a=b"的情况，其中"b"有名义值但不等于"a"，名义值可以被传播到其它的变量。] [对于 C89 中的外部函数，RealType 缺省映射为 double，IntegerType 缺省映射为 int。在 C99 标准附件 F 建议的映射中，RealType/double 与 IEC 60559:1989(ANSI/IEEE 754-1985)的双精度格式匹配。典型地，IntergerType 表示 32 位二进制补码的有符号整数。]*

3.7 内置变量 time(Built-in Variable time)

所有声明的变量都是自变量 **time** 的函数。time 是在所有类中都有效的内置变量，被视为输入变量。time 被隐式地定义为：

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

time 的 **start** 属性值被设置为仿真开始的时间点。

[例：

```
encapsulated model SineSource
  import Modelica.Math.sin;
  connector OutPort = output Real;
  OutPort y = sin(time);     // 使用内置变量 time。
end SineSource;
```

]

4 混合 DAE 的数学描述(Mathematical Description of Hybrid DAEs)

本节讨论 Modelica 模型向适当数学描述形式的映射。

第一步，Modelica 翻译器通过下列步骤将层次式的 Modelica 模型转换为 Modelica 语句的“平坦”集，包含所有用到组件的方程和算法段：

- 展开所有的类定义(平坦化继承树)，为模型的每个实例增加展开类的方程和赋值语句。
- 用连接集对应的方程替换所有的连接语句(参见 3.3.8.2 节)。
- 将所有的算法段映射到方程集。
- 将所有的 when 子句映射到方程集(参见 3.3.4)。

作为上述转换过程的结果，得到一个包含微分、代数和离散方程的方程集，形式如下($\mathbf{v} := [\dot{\mathbf{x}}; \mathbf{x}; \mathbf{y}; \mathbf{t}; \mathbf{m}; \mathbf{pre}(\mathbf{m}); \mathbf{p}]$)：

$$(1a) \quad \mathbf{c} := \mathbf{f}_c(relation(\mathbf{v}))$$
$$(1b) \quad \mathbf{m} := \mathbf{f}_m(\mathbf{v}, \mathbf{c})$$
$$(1c) \quad \mathbf{0} = \mathbf{f}_x(\mathbf{v}, \mathbf{c})$$

其中，

p	声明为 parameter 或 constant 的 Modelica 变量，即，与时间无关的变量
t	Modelica 变量 time，独立的(实型)变量
x(t)	Modelica 实型变量，存在微分
$m(t_e)$	Modelica 离散的实型、布尔型、整型未知量。这些变量只在事件时刻 te 改变其值，pre(m)为当前事件发生之前瞬时 m 的值
y(t)	Modelica 实型变量，不属于其他任何类型(=代数变量)
$c(t_e)$	所有生成的 if-表达式的条件，包括转换之后的 when-子句，参见 3.3.4 节
relation(v)	包含变量 v_i 的一种关系，例如， $v_1 > v_2, v_3 \geq 0$

为简单起见，**noEvent**()和 **reinit**()操作符的特殊情况未包含在上述方程中，

下面不作讨论。

生成的方程集用于仿真和其它的分析活动。仿真意即求解一个初值问题，即必须为状态变量 **x** 提供初始值，参见 3.3.9 节。这些方程定义了一个 DAEs(Differential Algebraic Equations，微分代数方程组)，可能具有非连续性、可变的结构，以及/或者被一个离散事件系统控制。这种类型的系统称为混合 DAE(hybrid DAEs)。

仿真按照下列方式进行：

1. DAE (1c)采用数值积分方法求解。在这个阶段，if-和 when-子句中的条件 c，以及离散变量 m 保持不变。因此，(1c)是一个连续变量的连续函数，并且满足数值积分的最基本要求。

2. 积分期间，监测来自(1a)的所有关系。如果任何一个关系改变其值，则触发一个事件，即发生变化的精确时间点被确定，积分中止。如同 3.5 节所讨论的，那些仅依赖于时间的关系通常用一种特殊的方式来处理，因为他们允许预先确定下一个事件的时间点。

3. 在某个事件时刻，(1)为一个代数方程的混合集，用于求解实型、布尔型和整型未知量。

4. 处理完一个事件之后，积分在步骤 1 重新开始。

注意，条件 **c** 的值以及 **m**(所有离散的实型、布尔型和整型变量)的值都只在事件时刻改变，这些变量在连续积分期间保持不变。在每个事件时刻，确定离散变量 **m** 的新值和状态变量 **x** 新的初值。离散变量的改变可能刻画一种新的 DAE 结构，其中状态变量 **x** 的元素被抑制。换句话说，在事件时刻，通过抑制 DAE 适当的分量，该 DAE 的状态变量、代数变量和残量方程的数目可能被改变。为了方程的清晰性，这一点没有在(1)中通过附加的标号显式表示出来。

在某个事件时刻，包括初始事件，模型方程按照以下迭代过程重新初始化：

```
已知量：x, t, p
未知量：dx/dt, y, m, pre(m), c
// pre(m) = 事件发生前 m 的值
loop
    pre(m)固定，求解(1)得到未知量
    if m == pre(m) then break
    pre(m) := m
end loop
```

求解(1)以求未知量并非易事，因为该方程集不仅包含实型，也包含布尔型和整型未知量。通常，在第一步中，这些方程被排序，在很多情况下，这些布尔型和整型未知量只能通过前向求解序列计算出来。在某些情况下，那些保留的方程系统(例如针对理想二极管、库仑摩擦力元)必须采用特殊的算法进行求解。

由于通过“平坦化”Modelica 模型来构造方程，混合 DAE(1)包含庞大数目的稀疏方程。因此，(1)的直接仿真需要稀疏矩阵方法。然而，采用数值方法直接求解这些初始方程集既不可靠又效率低下。一个原因是，许多 Modelica 模型，

象机械模型，具有 2 或 3 的 DAE 指标，即，模型状态变量的总数小于子组件的状态变量数之和。这种情况下，每种直接数值方法都存在求解困难，如果减少积分器步长，数值条件变得更差，零步长会导致奇异性。另一个问题是理想元素的处理，例如理想二极管或库仑摩擦力元。这些元素导致产生同时具有实型和布尔型未知量的混合方程系统。需要采用特殊的算法来求解这样的系统。

概括地说，需要符号转换技术将(1)转换为能可靠地进行数值求解的方程集。最重要的是，应采用 Pantelides 算法对特定的一部分方程进行微分，以缩减指标。注意，使用 Pantelides 算法缩减(1c)的指标之后，可以使用显式积分方法求解(1c)，例如 Runge-Kutta 算法：在连续积分期间，积分器提供 x 和 t；然后，(1c)变成计算代数变量 y 和状态导数 d**x**/dt 的线性或非线性方程组，模型通过求解这些方程系统，将 d**x**/dt 返回给积分器。通常，(1c)只是这些未知量的线性方程系统，因此可以直接求解。这个过程对于通常使用显式单步方法的实时仿真特别有用。

5 单位表达式(Unit Expression)

除非另作说明，Modelica 单位表达式的语法和语义符合国际标准 ISO 31/0-1992《关于量纲、单位和符号的一般原则》和 ISO 1000-1992《SI 单位及其乘积以及其它特定单位的推荐用法》。然而遗憾的是，这两个标准和其它现有或正在形成的 ISO 标准都没有定义单位表达式的形式语法。Modelica 采用了现有的一些推荐用法。

Modelica 使用的单位表达式的语法举例："N.m", "kg.m/s2", "kg.m.s-2", "1/rad", "mm/s"。

5.1 单位表达式的语法(The syntax of unit expression)

```
unit_expression:
    unit_numerator [ "/" unit_denominator ]
unit_numerator:
    "1" | unit_factors | "(" unit_expression ")"
unit_denominator:
    unit_factor | "(" unit_expression ")"
```

无量纲的度量单位用"1"表示。ISO 标准没有定义乘法和除法之间的优先级。ISO 建议最多有一个除号，其中，"/"右边的表达式要么不含乘号，要么包括在圆括号中。也可以使用负的指数，例如，"J/(kg.K)"可以写作"J.kg-1.K-1"。

```
unit_factors: unit_factor [ unit_mulop unit_factors ]
unit_mulop: "."
```

ISO 标准允许省略乘法操作符。但是，Modelica 强化了 ISO 建议，在正式说明中将乘法操作符显式写出。例如，Modelica 不支持"Nm"作为"牛顿米"，而要求写作"N.m"。

ISO 乘法操作符的首选符号是略高于基线的"圆点"："·"。Modelica 支持 ISO 的另一选项"."，该符号是位于基准线上的普通"圆点"。

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

ISO 标准没有定义任何求幂的操作符。unit_factor 由 unit_operand 组成，后面可跟一个可选的有符号整数，该整数解释为指数。unit_operand 与可选的 unit_exponent 之间不能有空格。

```
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

unit_symbol 是一个字母字符串。Modelica 单位的基本支持应能识别 SI 单位制的基本单位和导出单位。有可能支持用户自定义的单位符号。在基本版本中不支持希腊字母，而必须写出全名，例如"Ohm"。

unit_operand 应首先解释为 unit_symbol，只有在不成功时才使用第二个选项，并且接受一个前缀操作数。unit_symbol 和可选的 unit_prefix 之间不能有空格。前缀的值依据 ISO 标准确定。字符"u"用作前缀 micro 的符号。

5.2 例子(Examples)

- 由于前缀不能独立使用，单位表达式"m"代表米而不是千分之一(10^{-3})。

  毫米使用"mm"，而平方米m^2写作"m2"。

- 表达式"mm2"代表$mm^2 = (10^{-3}m)^2 = 10^{-6}m^2$。注意求幂包括前缀。单位表达式"T"代表特斯拉(磁通量密度单位，Tesla)，但注意字母"T"也是前缀万亿(tera)的符号，取值10^{12}。

6 外 部 函 数 接 口 (External Function Inteface)

6.1 概述(Overview)

这里，术语函数(function)意指任意的外部程序，不管该程序是否有返回值，或是否通过输出参数返回结果(或者两种情况都是)。Modelica 外部函数调用接口提供下列支持：

- 支持 C 和 FORTRAN77 编写的外部函数。其他语言如 C++和 Fortran90 可能在将来支持。
- 从 Modelica 到目标语言和目标语言到 Modelica 的双向参数类型映射。
- 从 Modelica 到目标语言的标准库之间简单自然的类型转换规则。
- 针对外部函数的任意参数顺序处理。
- 传递数组到外部函数和从外部函数传入数组，其中维数大小作为显式的整数参数传递。
- 同时用作输入和输出的外部函数参数处理。

外部函数声明的格式如下：

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

外部函数声明中 public 部分的组件应声明为 **input** 或 **output**。*[这对其他任何函数都是一样。protected 部分的组件允许声明局部变量，用于临时存储。]*

声明中的 language_specification 目前必须是"builtin"、"C"或"FORTRAN 77"之一。除非外部语言被指定，否则假定为 C。

"builtin"声明只用于函数，并且定义为 Modelic 内置函数。外部函数对内置("builtin")函数的调用机制由具体实现来定义。

[例：

package Modelica
package Math

```
            function sin
                input Real x;
                output Real y;
                external "builtin";
            end sin;
        end Math;
    end Modelica;

    model UserModel
        parameter Real p=Modelica.Math.sin(2);
    end UserModel;
```

]

声明中的 external_function_call 说明允许被调用的函数原型与下面定义的缺省假定不一致的函数，同时给出了用于调用外部函数的名字。如果外部调用函数没有显式给出，那么函数名字设定与 Modelica 的名字相同。

参数表中的表达式种类仅允许为标识符、标量常数、用于数组的函数 size 和常量维数。必要时，注解用于向编译器传递附加的信息。目前所支持的注解仅为 arrayLayout，可以是"rowMajor"或"columnMajor"。

6.2 形参类型映射(Arguments Type Mapping)

外部函数形参以与 Modelica 声明相同的顺序进行声明，除非 在显式的外部函数调用中进行指定。保护变量(即临时变量)使用与输出变量相同的方式来传递，而常量与维数表达式的传递方式与输入变量相同。

6.2.1 简单类型(Simple Types)

简单类型形参对 C 的缺省映射如下：

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

形参为 size(..., ...)的形式除外，这种情况下对应的 C 类型为 size_t。

为便于 C 函数调用，字符串设为以 NUL 结尾(即以'\0'结尾)。当返回一个非常量字符串时，该字符串的内存使用函数"ModelicaAllocateString"进行分配(参参见 6.6 节)*[使用 malloc 是不合适的，因为 Modelica 仿真环境可能会有自己的分配策略，例如，针对函数的局部变量采用一种特殊的栈]*。外部函数调用返回之后，Modelica 环境负责管理 ModelicaAllocateString 分配的内存(例如，适时释放内存)。不允许访问在该外部函数的前一次调用中使用 ModelicaAllocateString 分配的内存。

简单类型形参对 FORTRAN 77 的缺省映射如下：

Modelica	FORTRAN77	
	Input	Output
Real	DOUBLE PRECISION	DOUBLE PRECISION
Integer	INTEGER	INTEGER
Boolean	LOGICAL	LOGICAL
Enumeration type	INTEGER	INTEGER

目前不支持向 FORTRAN 77 子程序/函数传递字符串。

用作形参的枚举类型在调用外部 C 函数时映射为类型 int，在调用外部 FORTRAN 函数时映射为 INTEGER。第 i 个枚举常量映射成从 1 开始的整型值 i。

返回值映射成枚举类型与之类似：整型值 1 映射为第一个枚举常量，2 映射为第二个枚举常量，等等。对于某个枚举类型，返回一个不能映射成已有的枚举常量的数值是错误的。

6.2.2 数组(Arrays)

如果外部声明中没有出现显式的函数调用，那么数组通过其地址进行传递，后面跟 n 个类型为 size_t 的形参，表示对应的数组维度大小，其中，n 为数组的维数。*[类型 size_t 为 C 语言的无符号整型。]*

调用 C 函数时数组缺省以行主顺序存储；调用 FORTRAN 77 函数时缺省以列主顺序存储。这些缺省方式可通过数组布局注解来改写。参见下面的例子。

下表给出了在缺少显式外部函数调用情况下，调用 C 函数时数组形参的映射方式。类型 T 允许是可以传递给 C 的任意简单类型(定义于 6.2.1 节)，或者是记录类型(定义于 6.2.3 节)，T 映射为各节中定义的类型 T'。

Modelica	C
	Input and Output
T[dim1]	T' *, size_t dim1

T[dim1, dim2]	T' *, size_t dim1, size_t dim2
T[dim1, ..., dimn]	T' *, size_t dim1, ..., size_t dimn

在缺少显式外部函数调用情况下，用于传递数组形参到 FORTRAN 77 函数与上面定义的到 C 的方法类似：首先是数组地址，其次是整型的数组维数。参见下表，类型 T 允许是可以传递给 FORTRAN 77 的任意简单类型(定义于 6.2.1 节)，T 映射为该节中定义的类型 T'。

Modelica	FORTRAN77
	Input and Output
T[dim1]	T' *, INTEGER dim1
T[dim1, dim2]	T' *, INTEGER dim1, INTEGER dim2
T[dim1, ..., dimn]	T' *, INTEGER dim1, ..., INTEGER dimn

[以下两个例子说明了数组形参到外部 C 和 FORTRAN 77 函数的缺省映射。

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

对应的 C 语言原型如下：

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

如果外部函数用 FORTRAN 77 编写，即：

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

那么对应的 FORTRAN 77 函数的缺省定义如下：

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

]

当出现外部函数的显式调用时，数组及其维数大小必须显式传递。

[这个例子说明了当缺省假设不符合时，数组如何显式传递到外部FORTRAN 77 函数。

```
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;
```

对应的FORTRAN 77 子程序应声明如下：

```
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
```

下面的例子说明了如何以列主顺序传递数组到C 函数。

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

这对应于如下的C 函数原型：

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

]

6.2.3 记录(Records)

只支持针对 C 语言的记录类型映射。Modelica 记录类型包含简单类型、其它记录元素，或者固定维数的数组，其映射方式如下：

- 记录类通过 C 的 struct 来表示。

- Modelica 记录的每个元素映射为对应的 C 表示。Modelica 记录类中的元素以与 C 的 struct 相同的顺序进行声明。
- 数组映射为对应的 C 数组，并采用缺省的，或者显式的 arrayLayout 指令规定的数组布局。
- 记录通过引用方式传递(即传递指向记录的指针)。

例：

```
record R                          struct R {
    Real x;                           double x;
    Integer y[10];      映射成        int y[10];
    Real z;                           double z;
end R;                            };
```

6.3 返回类型映射(Return Type Mapping)

如果只有单个输出参数且没有显式的外部函数调用，或者有方程形式的显式外部调用，而此时方程左边必须是输出参数之一，那么外部程序认为是有返回值的函数。函数返回类型的映射按下表的说明进行。作为返回值的数组的存储空间由调用程序进行分配，因此返回数组的维数在调用时必须是固定的。否则，外部函数被认为不返回任何内容，即实际是一个过程，或者是 C 的 void 函数。*[这种情况下，6.2 节中所述的形参类型映射按没有显式外部函数调用的方式进行。]*

对于 C 和 FORTRAN 77，返回类型的缺省映射如下：

Modelica	C	FORTRAN77
Real	double	DOUBLE PRECISION
Integer	int	INTEGER
Boolean	int	LOGICAL
String	const char*	不允许
T[dim1, ..., dimn]	不允许	不允许
Enumeration type	int	INTEGER
Record	参见 6.2.3 节	不允许

数组的元素类型 T 可以是 6.2.1 节中定义的任意简单类型，对于 C 还可以是 6.2.3 节中定义的记录类型。

6.4 别名(Aliasing)

外部函数中任何可能的别名由 Modelica 工具而非用户负责管理。外部函数不

允许在其内部改变输入参数(即使他们在函数结束之前被恢复)。

[例：

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

下面的 Modelica 函数：

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

在多数系统中可翻译为下面的 C 函数

```
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;
}
```

不允许外部函数修改输入参数的原因在于，确保输入参数可以被存储在静态内存中，以避免多余的拷贝操作(尤其是矩阵)。如果程序不满足该要求，外部接口必须拷贝输入参数到临时变量。这种情况很少见但会遇到，例如在一些 Lapack 实现中的 dormlq。在那些特殊的情况下，外部接口的开发者必须拷贝输入参数到临时变量。如果第一个输入参数在 myfoo 内部被修改，外部接口的设计者将不得不改变接口函数"foo"，如下：

```
function foo
    input Real x;
    protected Real xtemp:=x; // 使用临时变量，因 myfoo 改变了输入参数
    public input Real y;
    output Real z;
    external "FORTRAN 77" myfoo(xtemp, y, z);
end foo;
```

注意，我们针对 Fortran 程序讨论输入参数，即使 Fortran 77 没有正式使用

输入参数而且对函数禁止在任何形参之间使用别名(X3J3/90.4 的 15.9.3.6 节)。对于严格遵从标准并且不能处理输入变量之间别名的新的 Fortran 77 编译器(如果有的话)，Modelica 工具必须将 foo 的第一个调用翻译成

```
temp1 = a; /*  临时变量以消除别名  */
myfoo(&a, &temp1, &b);
```

Modelica 中函数 foo 的使用不受这些因素影响。]

6.5 例子(Examples)

6.5.1 输入参数,有函数值(Input parameters, function values)

[这里传给外部函数的所有参数都是输入参数。返回一个函数值。如果没有指定外部语言，缺省为"C"，如下：

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

对应到下列 C 的原型：

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

Modelica 中的调用示例：

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

翻译成 C 的调用：

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

6.5.2 输出参数任意放置，没有外部函数值(Arbitrary placement of output parameters, no function value)

下面例子中，外部函数调用显式给出，允许以与 Modelica 版本中不同的顺序传递参数。

```
function foo
    input Real x;
    input Integer y;
    output Real u1;
    output Integer u2;
```

```
        external "C" myfoo(x, u1, y, u2);
    end foo;
```

对应到下列 C 的原型：

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

Modelica 中的调用示例：

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

翻译成 C 的调用：

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

6.5.3 具有函数值和输出变量的外部函数(External function with both function value and output variable)

下面的外部函数返回两个结果：一个函数值和一个输出参数值。二者都被映射成 Modelica 输出参数。

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

对应到下列 C 的原型：

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

Modelica 中的调用示例：

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

翻译成 C 的调用：

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

]

6.6 工具函数(Utility Functions)

下列工具函数可在 C 编写的外部 Modelica 函数中调用，在 ModelicaUtilities.h 中定义。

ModelicaMessage	*void ModelicaMessage(const char* string)* 输出消息字符串(无格式控制)。
ModelicaFormatMessage	*void ModelicaFormatMessage(const char* string, ...)* 输出消息，格式控制与 C 函数 printf 相同。
ModelicaError	*void ModelicaError(const char* string)* 输出错误消息字符串(无格式控制)。该函数不返回到调用函数，但是处理错误，类似于 Modelica 代码中的 assert。
ModelicaFormatError	*void ModelicaFormatError(const char* string, ...)* 输出错误消息，格式控制与 C 函数 printf 相同。该函数不返回到调用函数，但是处理错误，类似于 Modelica 代码中的 assert。
ModelicaAllocateString	*char* ModelicaAllocateString(size_t len)* 分配 Modelica 字符串的内存，用作外部 Modelica 函数的返回参数。注意，字符串数组(=字符串数组的指针)与其他数组一样，其存贮空间仍然由调用程序提供。如果发生错误，该函数不返回，而是调用"ModelicaError"。
ModelicaAllocateStringWithErrorReturn	*char* ModelicaAllocateStringWithErrorReturn(size_t len)* 除了在错误情况返回 0，该函数与 ModelicaAllocateString 相同。这允许外部函数在错误情况下关闭文件，并释放其他打开的资源。清理完毕资源之后，使用 ModelicaError 或 ModelicaFormatError 发出错误信号。

6.7 外部对象(External Objects)

外部函数在两次函数调用之间可具有内存。Modelica 中这种内存按下列规则定义为的预定义类型 ExternalObject 的实例：

- "ExternalObject"为预定义的抽象类。
  [由于该类是抽象的，不可能定义该类的实例。]
- 外部对象类应直接继承自"ExternalObject"，有且只有两个函数定义，分别称为"constructor"和"destructor"，并且不应包含其他元素。
- 在第一次使用该对象之前，constructor 函数只应调用一次。对每个完成构造的对象，在该对象最后一次使用之后，destructor 函数应只调用一次，即使有错误出现。constructor 有且只有一个输出参数，用于返回构造的 ExternalObject。Destructor 不应有输出参数，其唯一的输入参数是上次构造的 ExternalObject。显式调用 constructor 或 destructor 函数是非法的。
- 派生自 ExternalObject 的类既不能用于继承子句也不能用于简短类定义。

- 可以定义对 ExternalObject 内存进行操作的外部函数。用作外部 C 函数的输入参数或返回值的 ExternalObject 映射为 C 类型"void*"。

[例：

一个用户自定义表格可按以下方式定义为 ExternalObject(该表格以一种用户自定义格式从文件中读取，具有用于最后使用的表区间的内存)：

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

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

按以下方式使用：

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

需要提供下面的 Modelica 函数：

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

外部 C 函数可以按以下方式定义：

```
typedef struct { /* User-defined datastructure of the table */
    double* array; /* nrow*ncolumn vector */
    int nrow; /* number of rows */
    int ncol; /* number of columns */
    int type; /* interpolation type */
```

```
    int lastIndex; /* last row index for search */
} MyTable;

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

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

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

]

7 注解(Annotations)

注解用于存储模型相关的额外信息，例如图形、文档、版本等。Modelica 工具可以自由地定义和使用此处定义之外的其它注解。唯一的要求是，Modelica 工具必须能够完整地保存带有所有注解的文件，包括那些未被使用的。为确保如此，注解必须表示为符合 Modelica 语法结构。本文档规范定义了 Modelica 工具实现时应遵循的注解语义。

7.1 文档注解(Annotations for documentation)

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

"Documentation"注解包含给出文字说明的"info"注解、给出修订列表的"revisions"注解("revisions"可在打印文档中省略)，以及其他由 Modelica 工具定义的注解。

Modelica 工具如何解释"Documentation"注解中的信息未作规定。"Documentation"注解的字符串中，标记符<HTML>和</HTML>，或<html>和</html>可选地定义了 HTML 编码内容的开始和结束。到 Modelica 类的链接可以使用带有"Modelica"标记的 HTML 链接命令来定义。

MultiBody.Tutorial

与 Modelica 标记一起，URI 标识符#diagram、#info、#text、#icon 用于注明链接到模型的不同层次。例如：

Revolute

7.2 图形对象注解(Annotations for graphical objects)

Modelica 类的图形表示由两个抽象层次组成，即 icon 层和显示图形对象、组件图标、连接器和连接线的 diagram 层。icon 表示通过隐藏层次结构细节，典型地显示了组件外观。层次式的分解在 diagram 层描述，显示了子组件的图标。

本章描述的图形注解关联到下列 Modelica 语法。

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

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

层次描述(语法描述的开头)：

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

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

[例：

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

]

graphics 定义为有序的基本图元序列，如下所述*[注意，尽管 Modelica 没有定义不同类型对象数组的机制，这种有序序列在语法上是有效的 Modelica 注解]*。

7.2.1 共性定义(Common definifitions)

下列共性定义用于定义后续章节中的图形注解。

```
type DrawingUnit = Real(final unit="mm");
type Point = DrawingUnit[2] "{x, y}";
type Extent = Point[2] "Defines a rectangular area {{x1, y1}, {x2, y2}}";
```

"unit"的解释相对于打印机的正常尺寸输出(不缩放)。*[在屏幕上，正常尺寸的 1mm 典型地映射为 4 个像素。]*

所有图形实体都具有 visible 属性，指明该实体是否应显示。

```
partial record GraphicItem
    Boolean visible = true;
end GraphicItem;
```

7.2.1.1 坐标系(Coordinate systems)

每一层都有自己的坐标系。坐标系由左下角和右上角两点的坐标来定义。

```
record CoordinateSystem "Attribute to layer"
    Extent extent;
end CoordinateSystem;
```

[例如 icon 层的坐标系定义如下：

```
CoordinateSystem(extent = {{-10, -10}, {10, 10}});
```

即宽 20 单位、高 20 单位的第一象限坐标系。单位的精确解释一定程度上与 Modelica 工具相关。]

icon 和 diagram 的坐标系缺省定义如下；GraphicsItem 数组表示基本图元的有序链表。

```
record Icon "Representation of Icon layer"
    CoordinateSystem coordinateSystem(extent =
        {{-10, -10}, {10, 10}});
    GraphicItem[:] graphics;
end Icon;

record Diagram "Representation of Diagram layer"
    CoordinateSystem coordinateSystem(extent =
        {{-100, -100}, {100, 100}});
    GraphicItem[:] graphics;
end Diagram;
```

[例如一个连接器图标的坐标系可定义为：

```
CoordinateSystem(extent = {{-1, -1}, {1, 1}});
```

]

7.2.1.2 图形属性(Graphical properties)

图形对象和连接线的属性由下列属性类型来描述。

```
type Color = Integer[3](min=0, max=255) "RGB representation";

constant Color Black = zeros(3);

type LinePattern = enumeration(None,Solid,Dash,Dot,DashDot,DashDotDot);

type FillPattern = enumeration(None, Solid, Horizontal, Vertical,
    Cross, Forward, Backward, CrossDiag, HorizontalCylinder,
    VerticalCylinder, Sphere);

type BorderPattern = enumeration(None, Raised, Sunken, Engraved);
```

FillPattern 属性 Horizontal、Vertical、Cross、Forward、Backward 和 CrossDiag 指定填充模式，绘制时边界颜色在填充颜色之上。

属性 HorizontalCylinder、VerticalCylinder 和 Sphere 指定填充梯度，分别表示水平圆柱、垂直圆柱和球形。

边界模式属性 Raised 和 Sunken 表示一种面板，使用系统相关的方式来着色。

边界模式 Engraved 表示一种系统相关的轮廓。

```
type Arrow = enumeration(None, Open, Filled, Half);
```

```
type TextStyle = enumeration(Bold, Italic, UnderLine);
```

填充形状的边界和内部具有下列属性。

```
record FilledShape "Style attributes for filled shapes"
    Color lineColor = Black "Color of border line";
    Color fillColor = Black "Interior fill color";
    LinePattern pattern = LinePattern.Solid "Border line pattern";
    FillPattern fillPattern = FillPattern.None "Interior fill pattern";
    DrawingUnit lineThickness = 0.25 "Border line thickness"
end Style;
```

当指定颜色梯度时，采用指定颜色的色调和饱和度，颜色从指定的填充颜色褪色为白色和黑色。

7.2.2 组件实例与继承子句(Component instance and extends clause)

组件实例和继承子句可放入 diagram 或 icon 层。有一个带有 Placement 变型的注解来描述其位置，Placement 借助坐标系变换来定义。

```
record Transformation
    DrawingUnit x=0, y=0;
    Real scale=1, aspectRatio=1;
    Boolean flipHorizontal=false, flipVertical=false;
    Real rotation(quantity="angle", unit="deg")=0;
end Transformation;
```

坐标{x, y}定义组件图标坐标系的原点位置。当组件在其父类坐标系绘制时，scale 属性定义了 icon 坐标系的均匀比例因子。aspectRatio 属性定义非均匀缩放，其中 sy=aspectRatio*sx。flipHorizontal 和 flipVertical 属性定义绕坐标轴的水平和垂直翻转。绘图操作按照缩放、翻转和旋转的顺序发生作用。

```
record Placement "Attribute for component and extends"
    extends GraphicItem;
    Transformation transformation;
    Transformation iconTransformation "Placement in icon layer";
end Placement;
```

连接器可显示于模型的 icon 和 diagram 层。由于坐标系一般是不同的，位置信息需要用两个不同的坐标系给出。抽象视图可能需要连接器显示在不同位置，因此需要缩放和平移之外更多的灵活性。Placement 给出其在 diagram 层中的位置，iconPlacement 给出其在 icon 层中的位置。当连接器显示于 diagram 层时，其 diagram 层可见，以便打开层次化的连接器，允许连接到其中的内部连接器。

对于连接器，icon 层用于表现其在父模型的 icon 层显示时的效果。连接器的 diagram 层用于表现其在父模型的 diagram 层显示时的效果。Protected 连接器只

在 diagram 层显示。Public 连接器在 diagram 层和 icon 层都显示。非连接器组件只在 diagram 层显示。

7.2.3 连接(Connections)

连接用包含 Line 图元的注解进行表示，如下所示。

[例:

```
connect(a.x, b.x)
    annotation(Line(points={{-25,30}, {10,30}, {10, -20}, {40,-20}}));
```

]

7.2.4 基本图元(Graphical primitives)

本节描述用于定义注解中图形对象的基本图元。

7.2.4.1 线段(Line)

线段的定义如下：

```
record Line
  extends GraphicItem;

  Point points[:];

  Color color = Black;
  LinePattern pattern = LinePattern.Solid;
  DrawingUnit thickness = 0.25;

  Arrow arrow[2] = {Arrow.None, Arrow.None}; "{start arrow, end arrow}"
  DrawingUnit arrowSize=3;
  Boolean smooth = false "Spline";
end Line;
```

注意，Line 也用于定义连接的图形表示。

7.2.4.2 多边形(Polygon)

多边形的定义如下：

```
record Polygon
  extends GraphicItem;
```

```
    extends FilledShape;
    Point points[:];
    Boolean smooth = false "Spline outline";
  end Polygon;
```

如果第一点和最后一点不共点，多边形自动封闭。

7.2.4.3 矩形(Rectangle)

矩形的定义如下：

```
  record Rectangle
    extends GraphicItem;
    extends FilledShape;
    BorderPattern borderPattern = BorderPattern.None;
    Extent extent;
    DrawingUnit radius = 0 "Corner radius";
  end Rectangle;
```

extent 指定矩形的包围盒。如果 radius 指定，那么矩形用给定半径的圆角来绘制。

7.2.4.4 椭圆(Ellipse)

椭圆的定义如下：

```
  record Ellipse
    extends GraphicItem;
    extends FilledShape;
    Extent extent;
  end Ellipse;
```

extent 指定椭圆的包围盒。

7.2.4.5 文字(Text)

文本字符串的定义如下：

```
  record Text
    extends GraphicItem;
    extends FilledShape;
    Extent extent;
    String textString;
    DrawingUnit fontSize;
    String fontName;
```

```
    TextStyle textStyle[:];
  end Text;
```

样式属性 fontSize 指定字体大小。如果属性 fontSize 为 0，文字被缩放到适应其范围。否则指定为绝对大小。*[注意：单位"point"为 1/72 英寸，近似为 0.35 mm。]*

样式属性 textStyle 指定字体的变化。

7.2.4.6 位图(Bitmap)

位图图片的定义如下：

```
  record BitMap
    extends GraphicItem;
    Extent extent;
    String fileName "Name of bitmap file";
    String imageSource "Pixmap representation of bitmap";
  end BitMap;
```

位图图元描述可视的位图图片。图像数据可以存储于外部文件，或注解自身之中。图片被缩放到适应其范围。

当属性 fileName 指定时，该字符串引用到一个包含图片数据的外部文件。从字符串到文件的映射未作规定。支持的文件格式包括 PNG、BMP 和 JPEG，其他支持的文件格式未作规定。

当属性 imageSource 指定时，该字符串包含图片数据。图片表示为 Pixmap 格式。*[注意：Pixmap 格式定义是明确的，可用作文件格式，也可嵌套为字符串。已有开源的库用于读写 Pixmap 文件。]*

图片是均匀缩放的[以保持纵横比]，因此精确地适应其范围[沿一根轴到达其边界]。图片中心放在 extent 的中央。

7.3 图形用户界面注解(Annotations for the graphical user interface)

模型具有下列用于定义图形用户界面属性的注解。

- **annotation**(defaultComponentName = "name")
  生成给定类的组件时，推荐的组件名字为 name。
- **annotation**(defaultComponentPrefixes = "prefixes")
  生成组件时，推荐生成的声明形式为：
  prefixes class-name component-name
  下列前缀可包含于字符串 prefixes：inner、outer、replaceable、constant、

parameter、discrete。*[结合 defaultComponentName，使得用户易于生成与 outer 声明匹配的 inner 组件；参见下面的例子。]*

- **annotation**(missingInnerMessage = "message")
  当模型的 outer 组件没有对应的 inner 组件时，字符串 message 用作诊断消息。

[例：

```
model World
  annotation(defaultComponentName = "world",
             defaultComponentPrefixes = "inner replaceable",
             missingInnerMessage = "The World object is missing");
  ...
end World;
```

当模型 World 的实例被拖入 diagram 层时，生成下列声明：

```
inner replaceable World world;
```

]

声明可具有下列注解：

- **annotation**(unassignedMessage = "message");
  当变量声明中附有该注解时，如果由于方程结构的原因不能计算，字符串 message 可用作诊断消息。*[当采用 BLT 分解时，如果变量 a 或其别名 "b = a"、"b = -a" 不能赋值，那么该消息被显示出来。该注解用于提供库相关的错误消息。]*

- **annotation**(Dialog(enable = parameter-expression,
  tab = "tab", group = "group"));
  定义了组件或类参数在参数对话框中的位置，带有可选的 tab 和 group 说明。如果 enable 为 **false**，输入框被禁用*[并且不能给出输入]*。"Dialog" 定义为：

```
record Dialog
  parameter String tab = "General";
  parameter String group = "Parameters";
  parameter Boolean enable = true;
end Dialog;
```

参数对话框为一个 tab 序列，其中含有 group 序列。

[例：

```
connector Frame "Frame of a mechanical system"
    ...
    flow Modelica.SIunits.Force f[3]
    annotation(unassignedMessage = "
All Forces cannot be uniquely calculated. The reason could be that
```

```
    the mechanism contains a planar loop or that joints constrain the
    same motion. For planar loops, use in one revolute joint per loop the
    option PlanarCutJoint=true in the Advanced menu.
    ");
    end Frame;

    model BodyShape
        ...
        parameter Boolean animation = true;
        parameter SI.Length length "Length of shape"
        annotation(Dialog(enable = animation, tab = "Animation",
                          group = "Shape definition"));
        ...
    end BodyShape;
]
```

7.4 版本处理注解(Annotations for version handling)

顶层包或模型可指定其使用的顶层类的版本，其自身的版本号，如果可能还指定如何从以前版本进行转换。基于此，Modelica 工具可保证使用一致的版本，如果可能还从以前版本到当前版本升级模型用法。

7.4.1 版本号(Version numbering)

版本号的形式为：

- 主发行版本："""UNSIGNED_INTEGER{"."UNSIGNED_INTEGER}"""
  [例："2.1"]
- 预发行版本："""UNSIGNED_INTEGER{"."UNSIGNED_INTEGER}" "{S-CHAR} """
  [例："2.1 Beta 1"]
- 无序号版本："""NON-DIGIT{S-CHAR}"""
  [例："Test 1"]

主版本号是有序的，采用层次式的数字命名，后面跟对应的预发行版本号。与主发行版相同的预发行版本内部按字母顺序编号。

7.4.2 版本处理(Version handling)

在顶层类中，其版本号及其对于早期版本的依赖用一个或多个下面的注解来定义：

- version = CURRENT-VERSION-NUMBER
  定义模型或包的版本号。该顶层类中的所有类都具有该版本号。
- conversion (noneFromVersion = VERSION-NUMBER)
  定义使用 VERSION-NUMBER 版本的用户模型可不作任何改动地升级到当前类的 CURRENT-VERSION-NUMBER 版本。
- conversion (from (version = VERSION-NUMBER, script = "..."))
  定义使用 VERSION-NUMBER 版本的用户模型可用给定脚本升级到当前类的CURRENT-VERSION-NUMBER版本。转换脚本的语义未作定义。
- uses(IDENT (version = VERSION-NUMBER))
  定义该顶层类的模型使用了 VERSION-NUMBER 版本的顶层类 IDENT 中的模型。

注解 uses 和 conversions 可以包含多个不同的子实体。

[例：

```
package Modelica
  annotation(version=”2.1”,
             conversion(noneFromVersion=”2.1 Beta 1”,
                        from(version=”1.5”,
                             script=”convertFromModelica1_5.mos”)));
  ...
end Modelica;

model A
  annotation(version="1.0",
       uses(Modelica(version="1.5")));
  ...
end A;

model B
  annotation(uses(Modelica(version="2.1 Beta 1")));
  ...
end B;
```

该例中，模型A 使用了Modelica 库的一个老版本，可以用给定的脚本升级，模型B 使用了Modelica 库的一个老版本，但在升级时不需要进行改动。

]

7.4.3 映射版本到文件系统(Mapping of version to file system)

具有版本 VERSION-NUMBER 的顶层类 IDENT，可用下列方式之一保存到

MODELICAPATH 给定的目录中。

- 文件 IDENT ".mo" *[例：Modelica.mo]*
- 文件 IDENT " " VERSION-NUMBER ".mo" *[例：Modelica 2.1.mo]*
- 目录 IDENT *[例：Modelica]*
- 目录 IDENT " " VERSION-NUMBER *[例：Modelica 2.1]*

这允许 Modelica 工具使用同一个包的多个版本。

8 Modelica 标准库 (Modelica Standard Library)

预定义的免费"**package** Modelica"与 Modelica 翻译器一起提供。该库是一个范围广泛的多领域预定义组件的标准库。为改变模型设计过程的琐碎步骤，该库还包含一个标准的 **type** 和 **interface** 定义集。建模工作中，如果标准量纲的类型和连接器尽可能地依赖于标准库，那么模型兼容性及其重用性将得到加强。实现模型兼容性而不必采取建模活动之外的协调，对于建立全局可访问的模型库是必需的。自然地，不要求建模者使用标准库，并可增加任意数目的局部基础定义。

该库将作为正常的语言修订期间的一部分进行改进和修订。希望非正式的标准基础类在不同的领域得到发展，并逐步合并到 Modelica 标准库中。

库中的类型定义基于 ISO 31-1992。几个 ISO 量纲具有很长的名字，在实际建模工作中变得难以使用。因此，如必要的话也提供了较短的别名。例如在模型中反复使用"ElectricPotential"会变得很麻烦，因而提供了"Voltage"作为一种替换。

标准库不限于仅仅使用 SI 单位，只要通常的工程实践使用不同的(可能不兼容的)单位集，对应的量纲就允许出现于标准库，例如英制单位。为改善模型方程的状态，或保持实际值在一个更易于读写的数值附近，使用换算的 SI 单位书写模型也非常普遍。

连接器和抽象模型具有预定义的图形属性，使得他们的基本可视外观在所有基于 Modelica 的系统中是相同的。

完整的 Modelica 包可以从 http://www.Modelica.org/library/library.html 下载。下面给出了该库的介绍文档。注意，Modelica 包仍在开发中。

package Modelica
annotation(Documentation(info="
/* The Modelica package is a standardized, pre-defined and **free** package, that is shipped together with a Modelica translator. The package provides constants, types, connectors, partial models and model components in various disciplines.

In the Modelica package the following conventions are used:

- Class and instance names are written in upper and lower case letters, e.g., "ElectricCurrent". An underscore is only used at the end of a name to characterize a lower or upper index, e.g., body_low_up.

- Type names start always with an upper case letter.
Instance names start always with a lower case letter with only a few exceptions, such as "T" for a temperature instance.

```
            - A package XXX has its interface definitions in subpackage
              XXX.Interface, e.g., Electrical.Interface.

            - Preferred instance names for connectors:
              p,n: positive and negative side of a partial model.
              a,b: side "a" and side "b" of a partial model
              (= connectors are completely equivalent).
            */
      "));
end Modelica;
```

9 修订历史(Revision History)

本节描述 Modelica 语言设计历史及其主要贡献者。本文档的当前版本可从 http://www.modelica.org/得到。

9.1 Modelica 2.2

Modelica 2.2 发布于 2005 年 2 月 2 日。Modelica 2.2 规范由 Hans Olsson、Michael Tiller 和 Martin Otter 编辑校订。

9.1.1 Modelica 语言 V2.2 贡献者(Contributors of Modelica language, version 2.2)

(略)

9.1.2 Modelica 2.2 主要变化(Main Changes in Modelica 2.2)

Modelica 2.2 主要变化如下：

- 条件组件声明忽略依赖参数表达式的组件声明。引用不再出现的组件的连接方程被忽略。
- 在重声明中，一部分原始声明自动被新的声明继承。通过不重复其中的公共部分，使得更容易写出声明，特别适用于那些必须相同的属性。
- 递归的 **inner/outer** 定义，定义层次结构 inner/outer 声明，相互之间能够通信：同时使用 inner 和 outer 前缀声明的元素，在概念上引入两个具有相同名字的声明，一个遵循 inner 规则，另一个遵循 outer 规则。
- Modelica 函数中的数组可用一种方便、安全的方式改变其大小(函数中声明的非输入数组组件，维数大小由冒号(:)指定，并且没有绑定赋值，可在函数中通过赋值到一个完全数组来改变大小)。
- 引入一种新的连接器类型，称为“可扩展连接器”。这种连接器对于连接的连接器的名字匹配没有那么严格的要求，可方便地用于那些之前需要可替换连接器的场合，主要应用领域之一是构造复杂系统的信号总线。
- 导数操作符 **der**(expr)可用表达式作为参数，而不之前只能用一个变量名，例如，**der**(m*h)解释为 **der**(m)*h + m*__der__(h)。
- 可将一个函数定义为另一个函数的偏导数，例如："function Gibbs_T = der(Gibbs, T)"是计算函数 Gibbs 对输入参数 T 的偏导数的函数。

- 除"C"或"FORTRAN 77"之外，外部函数具有新的属性"builtin"。"builtin"说明仅用于定义 Modelica 语言的内置函数。"builtin"函数的外部函数调用机制是由实现来定义的。

语言变化是向后兼容的。

9.2 Modelica 2.1

Modelica 2.1 发布于 2004 年 1 月 30 日。Modelica 2.1 规范由 Hans Olsson 和 Martin Otter 编辑校订。

9.2.1 Modelica 语言 V2.1 贡献者(Contributors of Modelica language, version 2.1)

(略)

9.2.2 Modelica 2.1 主要变化(Main Changes in Modelica 2.1)

Modelica 2.1 主要变化如下：

- 枚举数组与枚举数组下标(必要的，例如正在开发的 Electrical.Digital 库)。
- 连接到层次结构的连接器(必要的，例如总线的易于实现)。
- Modelica 函数的可选输出参数。实际的输入和/或输出参数是否存在可用新的内置函数 isPresent(..)查询。以前的内置函数和属性 enable 已删除。
- 通过自动继承基类的约束类型到变型，使得缺省约束类型更加有用。
- 增强的重声明，必要的，例如，正在开发的 Modelica.Media 库(例如"redeclare model name"或"model extends name (<modifications>)")。
- 超定连接器的处理(必要的，例如多体系统和电力系统)，包括新的内置 package—Connections，具有 Connections.branch、Connections.root、Connections.potentialRoot、Connections.isRoot 等操作符。
- 算法段 while 循环中的 break 语句。
- Modelica 函数中的 return 语句。
- 内置函数 String(..)，提供 Boolean、Integer、Real 和 Enumeration 类型的字符串表示。
- 内置函数 Integer(..)，提供 Enumeration 类型的整型表示。
- 内置函数 semiLinear(..)，定义了具有两个斜坡的特征以及一套符号变换规则，特别是当函数成为欠约束时(该函数用于正在开发的 Modelica Fluid

库，用数学上清晰的方式定义回流)。

- 更一般性的标识符，即在单引号中包含任意字符，例如'+'或'123.456#1'是有效的标识符。'x'与 x 是不同的标识符。这对于直接将产品标识映射为模型名，以及将常用的数字符号定义为枚举(例如'+'、'-'、'0'、'1')是有用的。
- 新的注解：
  - 用于库和模型的版本处理(version, uses, conversion)，
  - 用于修订记录(revisions)，
  - 用于将 Modelica 名字作为 HTML 文档文本中的链接，
  - 用于 GUI 中方便的 "inner" 声明 (defaultComponentName, defaultComponentPrefixes)，
  - 用于参数化的菜单结构(Dialog, enable, tab, group)，以及
  - 用于库专用的错误消息(missingInnerMessage, unassignedMessage)。
- 修正了语法和语义规范中的一些小错误。

语言变化是向后兼容的，除了引入的新关键字 break 和 return，新的内置 package Connections 和删除的内置函数和属性 enable。

9.3 Modelica 2.0

Modelica 2.0 发布于 2002 年 1 月 30 日，草案发布于 2001 年 11 月 18 日。Modelica 2.0 规范由 Hans Olsson 编辑校订。从 2001 年 11 月开始，Modelica 是 Modelica Association 持有的注册商标。

9.3.1 Modelica 语言 V2.0 贡献者(Contributors of Modelica language, version 2.0)

(略)

9.3.2 Modelica 2.0 主要变化(Main Changes in Modelica 2.0)

关于 Modelica 2.0 增强的详细描述参见论文：

- M. Otter, H. Olsson: New Features in Modelica 2.0. 2nd International Modelica Conference, March 18-19, DLR Oberpfaffenhofen, Proceedings, pp. 7.1-7.12, 2002. This paper can be downloaded from http://www.Modelica.org/Conference2002/papers/p01_Otter.pdf
- Mattsson S. E., Elmqvist H., Otter M., and Olsson H.: Initialization of Hybrid Differential-Algebraic Equations in Modelica 2.0. 2nd International Modelica Conference, March 18-19, DLR Oberpfaffenhofen, Proceedings,

pp. 9-15, 2002. This paper can be downloaded from http://www.Modelica.org/Conference2002/papers/p02_Mattsson.pdf

Modelica 2.0 主要变化如下：

- 关于初始化的完整规范，以便在执行某些操作(如仿真或线性化)之前，计算出现在模型中的所有变量的相容初始值。
- 明确了 Modelica 对象框图的图形外观，因而保证了模型拓扑信息的可移植性，改进了以前非正式的图形描述，例如分离的图标和组件位置。
- 枚举类型，允许以一种可理解的、安全和高效的方式定义选项和属性。
- 支持(可选的)状态选择的显式优先选择，以便建模者加入应用特定的知识，用于引导求解过程，例如实时仿真。
- 数组构造器和约简操作符中的迭代器，支持功能更强大的表达式，特别是在声明中，以避免不方便而且低效的局部函数定义。
- 支持 block 的通用形式，适用于标量和向量连接器、(自动)向量化 block 的连接、更简单的 input/output 连接器。这允许 Modelica 输入/输出 block 库的重大简化，例如，以后只有所有 block 的标量版本是必须提供的。而且，新的库组件可以更容易地合并在一起。
- 记录构造器，允许构造数据表模型库。
- 函数可混用位置和名字实参。可选的结果和缺省参数使得同一个函数同时适合于初学者和高级用户。
- 针对外部 C 函数的附加工具，外部 C 函数作为 Modelica 模型的接口，尤其支持外部函数返回字符串、外部函数具有内部的内存(例如：连接用户定义的表格、特性数据库、稀疏矩阵处理、硬件接口)。
- 增加了一个索引，以及一些以前没有正式定义的基本结构的规范，例如 while 子句、if 子句。

语言变化是向后兼容的，除了引入的新关键字 enumeration。block 库不久将变得可用，其中的变化要求用户模型作一些改变。

9.4 Modelica 1.4

Modelica 1.4 发布于 2000 年 11 月 15 日。Modelica Association 成立于 2000 年 2 月 5 日，现在负责 Modelica 语言的设计。Modelica 1.4 规范由 Hans Olsson 和 Dag Brück 编辑校订。

9.4.1 Modelica 语言 V1.4 贡献者(Contributors of Modelica language, version 1.4)

(略)

9.4.2 Modelica 标准库贡献者(Contributors of Modelica Standard Libaray)

(略)

9.4.3 Modelica 1.4 主要变化(Main Changes in Modelica 1.4)

- 去除了使用前声明的规则。这简化了图形用户环境，因为当组件被可视化地组合到一起时，其中不存在声明顺序。
- 引入 encapsulated 类和 import 机制，改进了 package 的概念。encapsulated 类可被视为“独立单元”：当拷贝或移动封装类时，该类中最多只有 import 语句必须改变。
- 改进的 when 子句。nondiscrete 关键字被去除，when 子句中的方程左边变量必须有唯一的变量名，并且定义了从 when 子句到方程的精确映射。结果是，when 子句现在不需引用一个排序算法就可以精确定义，而且有可能处理不同条件 when 子句之间的、以及 when 子句与模型中连续部分之间的代数环。discrete 关键字现在是可选的，这简化了模型库的开发，因为只有一种连接器类型是必需的，而不是在变量上包含或者不包含 discrete 前缀的多种类型。另外，算法段中的 when 子句可以有 elsewhen 子句，简化了 when 子句之间优先权的定义。
- 针对可替换的声明：允许约束子句以及适用于重声明的注解列表。这允许图形用户环境自动构造带有有意义选项的菜单。
- 函数可指定其导数。这允许应用 Pantelides 算法也可针对外部函数缩减 DAE 指标。
- 新的内置操作符"**rem**"(remainder，余数)。**rem** 和以前只允许用于 when 子句的内置操作符 **div、mod、ceil、floor、integer** 现在可用于任意地方，因为当这些操作符的结果值非连续地变化时，自动生成状态事件。
- Quantity 属性也可用于基本类型 Boolean、Integer、String(而不只用于 Real)，以便抽象变量可以引用物理量纲(例如 Boolean i(quantity="Current")，若电流正在流过，为真；若没有电流流过，为假)。
- final 关键字允许用于声明中，以便阻止其变型，例：

```
model A
    Real x[:];
    final Integer n = size(x, 1);
end A;
```

- 几个细微增强，例如在变型中使用点号(例如："A x(B.C=1, B.D = 2)"等同于"A x(B(C = 1, D = 2));")。
- 规范内部重新调整结构。

Modelica 1.4 与 Modelica 1.3 是向后兼容的，下列情况除外：(1)一些极其异常的情况，使用了已去除的“使用前声明规则”和以前的声明顺序，得到不同的结果；(2)方程段中的 when 子句，使用了一般形式"expr1 = expr2"(现在只有 v = expr 和函数的一些特殊情况是允许的)；(3)一些极其异常的情况，when 子句可能在初始化时不再计算，因为 when 条件的初始化现在用更有意义的方式来定义(在 Modelic 1.4 之前，when 子句中每个条件的"previous"值为 false)；(4)含有已去除的 nondiscrete 关键的模型。

9.5 Modelica 1.3 及更早版本

Modelica 1.3 发布于 1999 年 12 月 15 日。

9.5.1 截至 Modelica 1.3 之前的贡献者(Contributors up to Modelica 1.3)

(略)

9.5.2 Modelica 1.3 主要变化(Main Changes in Modelica 1.3)

Modelica 1.3 发布于 1999 年 12 月 15 日。

- 定义了 inner/outer 连接器的连接语义。
- 定义了 protected 元素的语义。
- 定义了具有最小可变性前缀的变量优先。
- 改进了数组表达式的语义定义。
- 定义了 for 循环变量的作用域。

9.5.3 Modelica 1.2 主要变化(Main Changes in Modelica 1.2)

Modelica 1.2 发布于 1999 年 6 月 15 日。

- 改变了外部函数接口，提供了更大的灵活性。
- 为动态类型引入了 inner/outer。
- 重新定义 final 关键字，仅仅限制进一步的变型。
- 限制重声明只针对可替换元素。
- 定义了 if 子句的语义。
- 定义了可用的代码优化。
- 改进了事件处理的语义。
- 引入 fixed 和 nominal 属性。
- 引入 terminate 和 analysisType。

9.5.4 Modelica 1.1 主要变化(Main Changes in Modelica 1.1)

Modelica 1.1 发布于 1998 年 12 月。

主要变化：

- 规范作为单独的文档，与基本原理分开。
- 引入前缀 discrete 和 nondiscrete。
- 引入 pre 和 when。
- 定义了数组表达式的语义。
- 引入内部函数和操作符(Modelica 1.0 中只出现了 connect)。

9.5.5 Modelica 1.0

Modelica 1.0 是 Modelica 的第一个版本，发布于 1997 年 9 月，具有了作为基本原理简短附录的语言规范。

10　索引(Index)

(略)