Arrays, Algorithms, and Functions

Array Data Structures

- An **array variable** is an ordered collection of scalar variables all of the same type
- **Dimensions** – Each array has a certain **number of dimensions**, a vector has 1 dimension, a matrix 2 dimensions
- Modelica arrays are "**rectangular**", i.e., all rows in a matrix have equal length, and all columns have equal length
- **Construction** – An array is **created** by declaring an array variable or calling an array constructor
- **Indexing** – An array can be **indexed** by Integer, Boolean, or enumeration values
- Lower/higher **bounds** – An array dimension indexed by integers has 1 as lower bound, and the size of the dimension as higher bound

Two forms of Array Variable Declarations

Array declarations with dimensions in the type

2 dimensions

```
Real[3]        positionvector = {1,2,3};
Real[3,3]      identitymatrix = {{1,0,0}, {0,1,0}, {0,0,1}};
Integer[n,m,k] arr3d;
Boolean[2]     truthvalues = {false,true};
Voltage[10]    voltagevector;
String[3]      str3 = {"one", "two", "blue"};
Real[0,3]      M;     // An empty matrix M
```

Array declarations with dimensions behind the variable

1 dimension

```
Real       positionvector[3] = {1,2,3};
Real       identitymatrix[3,3] = {{1,0,0}, {0,1,0}, {0,0,1}};
Real       arr3d[n,m,k];
Boolean    truthvalues[2] = {false,true};
Voltage    voltagevector[10]
String     str3[3] = {"one", "two", "blue"};
Integer    x[0];        // An empty vector x
```

Flexible Array Sizes

Arrays with unspecified dimension sizes are needed to make flexible models that adapt to different problem sizes

An unspecified Integer dimension size is stated by using a colon (:) instead of the usual dimension expression

Flexible array sizes can only be used in functions and partial models

```
Real[:,:]  Y;              // A matrix with two unknown dimension sizes
Real[:,3]  Y2;             // A matrix where the number of columns is known
Real       M[:,size(M,1)]  // A square matrix of unknown size

Real[:]  v1, v2;           // Vectors v1 and v2 have unknown sizes

Real[:]  v3 = fill(3.14, n);   // A vector v3 of size n filled with 3.14
```

Modifiers for Array Variables

Array variables can be initialized or given start values using modifiers

```
Real A3[2,2];                       // Array variable
Real A4[2,2](start={{1,0},{0,1}}); // Array with modifier
Real A5[2,2](unit={{"Voltage","Voltage"},{"Voltage","Voltage"}});
```

Modification of an indexed element of an array is illegal

```
Real A6[2,2](start[2,1]=2.3);
// Illegal! indexed element modification
```

The **each** operator can give compact initializations

```
record Crec
  Real b[4];
end Crec;
model B
  Crec A7[2,3](b = {
 {{1,2,3,4},{1,2,3,4},{1,2,3,4}},
 {{1,2,3,4},{1,2,3,4},{1,2,3,4}} } );
end B;
```

same A7 as

```
model C
  Crec A7[2,3](each b = {1,2,3,4});
end C;
```

Array Constructors { } and Range Vectors

An array constructor is just a function accepting scalar or array arguments and returning an array result. The array constructor function array(A,B,C,...), with short-hand notation {A, B, C, ...}, constructs an array from its arguments

```
{1,2,3}                     // A 3-vector of type Integer[3]
array(1.0,2.0,3)             // A 3-vector of type Real[3]
{{11,12,13}, {21,22,23}}    // A 2x3 matrix of type Integer[2,3]
{{{1.0, 2.0, 3.0}}}         // A 1x1x3 array of type Real[1,1,3]
{ {1}, {2,3} }              // Illegal, arguments have different size
```

Range vector constructor

Two forms: *startexpr* : *endexpr* or *startexpr* : *deltaexpr* : *endexpr*

```
Real v1[5] = 2.7 : 6.8;                  // v1 is {2.7, 3.7, 4.7, 5.7, 6.7}
Real v2[5] = {2.7, 3.7, 4.7, 5.7, 6.7}; // v2 is equal to v1
Integer v3[3] = 3 : 5;                   // v3 is {3,4,5}
Integer v4empty[0] = 3 : 2               // v4empty is an empty Integer vector
Real    v5[4]  = 1.0 : 2  : 8;           // v5 is {1.0,3.0,5.0,7.0}
Integer v6[5]  = 1   : -1 : -3;          // v6 is {1,0,-1,-2,-3}
```

Set Formers – Array Constructors with Iterators

Mathematical notation for creation of a set of expressions, e.g. $A = \{1, 3, 4, 6...\}$

$\{ expr_i \mid i\ \varepsilon\ A\}$ equivalent to $\{ expr_1 , expr_3 , expr_4 , expr_{6,} \ldots \}$

Modelica has array constructors with iterators, single or multiple iterators

$\{ expr_i$ **for** i **in** $A\}$ $\{ expr_{ij}$ **for** i in A, j **in** $B \}$

If only one iterator is present, the result is a vector of values constructed by evaluating the expression for each value of the iterator variable

```
{r for r in 1.0 : 1.5 : 5.5} // The vector 1.0:1.5:5.5={1.0, 2.5, 4.0, 5.5}
{i^2 for i in {1,3,7,6}}     // The vector {1, 9, 49, 36}
```

Multiple iterators in an array constructor

```
 {(1/(i+j-1) for i in 1:m, j in 1:n};        // Multiple iterators
{{(1/(i+j-1) for j in 1:n} for i in 1:m}     // Nested array constructors
```

Deduction of range expression, can leave out e.g. `1:size(x,1)`

```
Real s1[3] = {x[i]*3 for i in 1:size(x,1)};  // result is {6,3,12}
Real s2[3] = {x[i] for i};     // same result, range deduced to be 1:3
```

Array Concatenation and Construction

General array concatenation can be done through the array concatenation operator cat(k,A,B,C,...) that concatenates the arrays A,B,C,... along the kth dimension

```
cat(1, {1,2}, {10,12,13} )           // Result: {1,2,10,12,13}
cat(2, {{1,2},{3,4}}, {{10},(11}} )  // Result: {{1,2,10},{3,4,11}}
```

The common special cases of concatenation along the first and second dimensions are supported through the special syntax forms [A;B;C;...] and [A,B,C,...] respectively, that also can be mixed

Scalar and vector arguments to these special operators are promoted to become matrices before concatenation – this gives MATLAB compatibility

```
[1,2; 3,4]                      // Result: {{1,2}, {3,4}}
[ [1,2; 3,4], [10; 11] ]        // Result: [1,2,10; 3,4,11]
cat(2, [1,2; 3,4], [10; 11] )   // Same result: {{1,2,10}, {3,4,11}}
```

Array Indexing

The array indexing operator ...[...] is used to access array elements for retrieval of their values or for updating these values

***arrayname* [*indexexpr1*, *indexexpr2*, ...]**

```
Real[2,2] A = {{2,3},{4,5}};  // Definition of Array A
A[2,2]                        // Retrieves the array element value 5
A[1,2] := ...                 // Assignment to the array element A[1,2]
```

Arrays can be indexed by integers as in the above examples, or Booleans and enumeration values, or the special value **end**

```
type Sizes = enumeration(small, medium);
Real[Sizes]    sz = {1.2, 3.5};
Real[Boolean] bb = {2.4, 9.9};

sz[Sizes.small]     // Ok, gives value 1.2
bb[true]            // Ok, gives value 9.9

A[end-1,end]        // Same as:  A[size(A,1)-1,size(A,2)]
A[v[end],end]       // Same as:  A[v[size(v,1)],size(A,2)]
```

Array Addition, Subtraction, Equality and Assignment

Element-wise addition and subtraction of scalars and/or arrays can be done with the (+), (.+), and (-), (.-) operators. The operators (.+) and (.-) are defined for combinations of scalars with arrays, which is not the case for (+) and (-)

```
{1,2,3} + 1                      // Not allowed!
{1,2,3} – {1,2,0}                // Result: {0,0,3}
{1,2,3} + {1,2}                  // Not allowed, different array sizes!
{{1,1},{2,2}} + {{1,2},{3,4}}    // Result: {{2,3},{5,6}}
```

Element-wise addition (.+) and element-wise subtraction (.□)

```
{1,2,3} .+ 1                      // Result: {2,3,4}
1 .+ {1,2,3}                      // Result: {2,3,4}
{1,2,3} .– {1,2,0}                // Result: {0,0,3}
{1,2,3} .+ {1,2}                  // Not allowed, different array sizes!
{{1,1},{2,2}} .+ {{1,2},{3,4}}    // Result: {{2,3},{5,6}}
```

Equality (array equations), and array assignment

```
v1 = {1,2,3};
v2 := {4,5,6};
```

Array Multiplication

The linear algebra multiplication operator (*) is interpreted as scalar product or matrix multiplication depending on the dimensionality of its array arguments.

```
{1,2,3} * 2                       // Elementwise mult: {2,4,6}
3 * {1,2,3}                       // Elementwise mult: {3,6,9}
{1,2,3} * {1,2,2}                 // Scalar product:   11

{{1,2},{3,4}} * {1,2}             // Matrix mult:      {5,11}
{1,2,3} * {{1},{2},{10}}          // Matrix mult:      {35}
{1,2,3} * [1;2;10]                // Matrix mult:      {35}
```

Element-wise multiplication between scalars and/or arrays can be done with the (*) and (.*) operators. The (.*) operator is equivalent to (*) when both operands are scalars.

```
{1,2,3} .* 2                      // Result: {2,4,6}
2 .* {1,2,3}                      // Result: {2,4,6}
{2,4,6} .* {1,2,2}                // Result: {2,8,12}
{1,2,3} .* {1,2}                  // Not allowed, different array sizes!
```

Array Dimension and Size Functions

An array reduction function “reduces” an array to a scalar value, i.e., computes a scalar value from the array

ndims(A)	Returns the number of dimensions k of array A, with k >= 0.
size(A,i)	Returns the size of dimension i of array A where 1 <= i <= ndims(A).
size(A)	Returns a vector of length ndims(A) containing the dimension sizes of A.

```
Real[4,1,6] x;  // declared array x

ndims(x);          // returns 3 – no of dimensions
size(x,1);         // returns 4 – size of 1st dimension
size(x);           // returns the vector {4, 1, 6}
size(2*x+x) = size(x);  // this equation holds
```

Array Reduction Functions and Operators

An array reduction function "reduces" an array to a scalar value, i.e., computes a scalar value from the array. The following are defined:

min(A)	Returns the smallest element of array A.
max(A)	Returns the largest element of array A.
sum(A)	Returns the sum of all the elements of array A.
product(A)	Returns the product of all the elements of array A.

```
min({1,-1,7})                       // Gives the value -1
max([1,2,3; 4,5,6])                 // Gives the value 6
sum({{1,2,3},{4,5,6}})              // Gives the value 21
product({3.14, 2, 2})               // Gives the value 12.56
```

Reduction functions with iterators

```
min(i^2 for i in {1,3,7})     // min(min(1, 9), 49)) = 1
max(i^2 for i in {1,3,7})     // max(max(1, 9), 49)) = 49
sum(i^2 for i in {1,3,7,5})
```

Vectorization of Function Calls with Array Arguments

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

```
sin({a,b,c})      // Vector argument, result: {sin(a),sin(b),sin(c)}
sin([1,2; 3,4])   // Matrix arg, result: [sin(1),sin(2);sin(3),sin(4)]
```

Functions with more than one argument can be generalized/vectorized to elementwise application. All arguments must be the same size, traversal in parallel

```
atan2({a,b,c},{d,e,f})   // Result: {atan2(a,d), atan2(b,e), atan2(c,f)}
```

Algorithms and Statements

Algorithm Sections

Whereas equations are very well suited for physical modeling, there are situations where computations are more conveniently expressed as algorithms, i.e., sequences of instructions, also called statements

algorithm
  ...
  <statements>
  ...
<some keyword>

Algorithm sections can be embedded among equation sections

```
equation
  x = y*2;
  z = w;
algorithm
  x1 := z+x;
  x2 := y-5;
  x1 := x2+y;
equation
  u = x1+x2;
  ...
```

Iteration Using for-statements in Algorithm Sections

for *<iteration-variable>* **in** *<iteration-set-expression>* **loop**
  <statement1>
  <statement2>
  ...
end for

The general structure of a `for`-statement with a single iterator

```
class SumZ
  parameter Integer n = 5;
  Real[n]  z(start = {10,20,30,40,50});
  Real sum;
algorithm
  sum := 0;
  for i in 1:n loop      // 1:5 is {1,2,3,4,5}
    sum := sum + z[i];
  end for;
end SumZ;
```

A simple `for`-loop summing the five elements of the vector `z`, within the class `SumZ`

Examples of `for`-loop headers with different range expressions

```
for k in 1:10+2 loop          // k takes the values 1,2,3,...,12
for i in {1,3,6,7} loop       // i takes the values 1, 3, 6, 7
for r in 1.0 : 1.5 : 5.5 loop // r takes the values 1.0, 2.5, 4.0, 5.5
```

Iterations Using while-statements in Algorithm Sections

while *<conditions>* **loop**
  <statements>
end while;

The general structure of a `while`-loop with a single iterator.

```
class SumSeries
  parameter Real eps = 1.E-6;
  Integer i;
  Real sum;
  Real delta;
algorithm
  i := 1;
  delta := exp(-0.01*i);
  while delta>=eps loop
    sum := sum + delta;
    i := i+1;
    delta := exp(-0.01*i);
  end while;
end SumSeries;
```

The example class `SumSeries` shows the `while`-loop construct used for summing a series of exponential terms until the loop condition is violated , i.e., the terms become smaller than eps.

`if`-statements

```
if <condition> then
   <statements>
elseif <condition> then
   <statements>
else
   <statementss>
end if
```

The general structure of `if`-statements. The `elseif`-part is optional and can occur zero or more times whereas the optional `else`-part can occur at most once

```
class SumVector
  Real sum;
  parameter Real v[5] = {100,200,-300,400,500};
  parameter Integer n = size(v,1);
algorithm
  sum := 0;
  for i in 1:n loop
    if v[i]>0 then
      sum := sum + v[i];
    elseif v[i] > -1 then
      sum := sum + v[i] -1;
    else
      sum := sum - v[i];
    end if;
  end for;
end SumVector;
```

The `if`-statements used in the class `SumVector` perform a combined summation and computation on a vector `v`.

when-statements

```
when <conditions> then
  <statements>
elsewhen <conditions> then
  <statements>
end when;
```

when-statements are used to express actions (statements) that are only executed at events, e.g. at discontinuities or when certain conditions become true

There are situations where several assignment statements within the same when-statement is convenient

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

```
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;
```

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

Algorithm and equation sections can be interleaved.

Functions

Function Declaration

The structure of a typical function declaration is as follows:

```
function <functionname>
  input  TypeI1 in1;
  input  TypeI2 in2;
  input  TypeI3 in3;
  ...
  output TypeO1 out1;
  output TypeO2 out2;
  ...
protected
  <local variables>
  ...
algorithm
  ...
  <statements>
  ...
end <functionname>;
```

All internal parts of a function are optional, the following is also a legal function:

```
function <functionname>
end <functionname>;
```

Modelica functions are *declarative mathematical functions*:

- Always return the same result(s) given the same input argument values

Function Call

Two basic forms of arguments in Modelica function calls:

- *Positional* association of actual arguments to formal parameters
- *Named* association of actual arguments to formal parameters

Example function called on next page:

```
function PolynomialEvaluator
  input  Real A[:];    // array, size defined
                       // at function call time
  input  Real x := 1.0;// default value 1.0 for x
  output Real sum;
protected
  Real   xpower;          // local variable xpower
algorithm
  sum := 0;
  xpower := 1;
  for i in 1:size(A,1) loop
    sum := sum + A[i]*xpower;
    xpower := xpower*x;
  end for;
end PolynomialEvaluator;
```

The function `PolynomialEvaluator` computes the value of a polynomial given two arguments:
a coefficient vector `A` and a value of `x`.

Positional and Named Argument Association

Using *positional* association, in the call below the actual argument `{1,2,3,4}` becomes the value of the coefficient vector `A`, and `21` becomes the value of the formal parameter `x`.

```
...
algorithm
  ...
  p:= polynomialEvaluator({1,2,3,4},21)
```

The same call to the function `polynomialEvaluator` can instead be made using *named* association of actual parameters to formal parameters.

```
...
algorithm
  ...
  p:= polynomialEvaluator(A={1,2,3,4},x=21)
```

Functions with Multiple Results

```
function PointOnCircle"Computes cartesian coordinates of point"
  input  Real angle  "angle in radians";
  input  Real radius;
  output Real x;    // 1:st result formal parameter
  output Real y;    // 2:nd result formal parameter
algorithm
  x := radius * cos(phi);
  y := radius * sin(phi);
end PointOnCircle;
```

Example calls:

```
(out1,out2,out3,...) = function_name(in1, in2, in3, in4, ...);  // Equation
(out1,out2,out3,...) := function_name(in1, in2, in3, in4, ...); // Statement

(px,py)  = PointOnCircle(1.2, 2);   // Equation form
(px,py) := PointOnCircle(1.2, 2);   // Statement form
```

Any kind of variable of compatible type is allowed in the parenthesized list on the left hand side, e.g. even array elements:

```
(arr[1],arr[2]) := PointOnCircle(1.2, 2);
```

External Functions

It is possible to call functions defined outside the Modelica language, implemented in C or FORTRAN 77

```
function polynomialMultiply
  input  Real a[:], b[:];
  output Real c[:] := zeros(size(a,1)+size(b, 1) - 1);
external
end polynomialMultiply;
```

The body of an external function is marked with the keyword **`external`**

If no language is specified, the implementation language for the external function is assumed to be C. The external function `polynomialMultiply` can also be specified, e.g. via a mapping to a FORTRAN 77 function:

```
function polynomialMultiply
  input  Real a[:], b[:];
  output Real c[:] := zeros(size(a,1)+size(b, 1) - 1);
external ”FORTRAN 77”
end polynomialMultiply;
```

- Arrays Algorithms Functions and Exercises.onb