



Language Reference Manual 6

All contents copyright © 2008 Tizio BV, Netherlands Environmental Assessment Agency.

MyM Language Reference Manual 6.1 for Windows ®.

All rights reserved. No part of this document or the related files may be reproduced or transmitted in any form, by any means (electronic, photocopying, recording, or otherwise) without the prior written permission of the publisher.

Although care has been taken in preparing the information contained in this site, all material that is provided here or could be reached by using the website as a starting point is supplied “as is” without warranty of quality or accuracy or of any other kind. Neither Tizio B.V, Netherlands Environmental Assessment Agency, nor any author or supplier contributing to this manual is responsible for any errors or omissions in any information provided or the results obtained from the use of such information.

website: www.my-m.eu

e-mail: info@my-m.eu

Contents

1. Introduction	3	7. Equations	18
Lexical conventions	3	Dependent variables	19
Comments	3	Conversion	19
Identifiers	3	8. Expressions	20
Keywords	4	Primary expression	21
Constants	4	9. Loops	22
Operators	4	Implicit loops	22
File inclusion	4	Explicit loops	23
Macros	5	10. Functions	25
Syntax notation	5	Mathematical functions	25
2. Mathematical model	6	User functions	26
3. Constant definitions	8	Differential equations	29
4. Variables	9	Loop functions	32
Type	9	Conditional functions	33
Size	9	External functions	34
Use	10	Void functions	37
Domain dependency	11	11. Output equations	38
Variable dependency	12	12. Modules	39
Scope	12	13. Processing	42
5. Variable declarations	13	14. References	44
6. Domain specifications	16		

1. Introduction

In this manual the language for the definition of mathematical models within MyM is described. The language was designed to resemble the standard mathematical notation. Many options are provided to make these equations as compact as possible.

The purpose of this document is to provide a detailed description of the features of the language. For a tutorial see the [Language Tutorial].

Lexical conventions

On the lexical level a MyM model is a sequence of tokens: identifiers, keywords, constants, operators, and separators.

Blanks, tabs, newlines, and comments are ignored except as they serve to separate tokens. No distinction is made between upper and lower case letters.

Comments

All input on a line after an exclamation mark (!) is considered as comment and ignored.

Identifiers

An identifier is a letter, followed by a sequence of letters, digits, and underscores. An identifier references to a module, constant, a variable, or a function. The maximum length of an identifier is 64 characters. The following identifiers are predefined:

```
variables:  t, i, j, k;  
constants: true, false, RK1, RK2, RK4, RKV;  
functions: section "Functions" on page 25.
```

Keywords

The following identifiers are used as keywords, and may not be used otherwise:

```
real  double  integer  boolean  time  void  const
step  to      downto   switch   else
module begin  end      import  export
extern file   prog
```

Constants

An integer constant is a sequence of digits, optionally preceded by a plus or minus sign. A real constant consists of an integer part, a decimal point, a fraction part, an *e* or *E*, followed by an optionally signed integer exponent. The integer part and the fraction part are sequences of digits. All parts are optional, except the integer part and the decimal point.

Operators

The following characters and character combinations are recognized as operators, or have other special meaning within the MyM language:

and	or	not		
=>	>=	<	<=	<>
+	-	*	/	**
()	[]	
;	,	?	.	

File inclusion

If the first character of a line is a *#*, followed by the directive include and a filename, then this file is included during the processing of the programme.

The include directive may be nested: used in included files again. Recursion is obviously not allowed, and is detected. Example:

```
#include submodel.m
```

Macros

Macros can be used for the symbolic definition of multiple occurring fragments of text. They are similar to macros as defined for the C programming language. Their form is:

```
#define name replacement text
```

Again, the # should be the first character of the line. Subsequent occurrences of *name* are replaced by replacement text. The name has the same form as an identifier; the replacement text is arbitrary.

Normally the replacement text is the rest of the line, but a long text can be spread over multiple lines by hiding newlines with a backslash (' \ '). The scope of a macro is from its point of definition to the end of the model. However, a macro can be removed by the following command:

```
#undef name
```

A definition may use previous definitions. It is also possible to define macros with arguments, so the replacement text can be different for different calls. As an example, consider:

```
#define square(x) (x)*(x)
```

Note that the expansion of arguments is purely textual, macros should not be confused with functions. If the parentheses in the preceding macro definition are not used, then the effect of invocation of square(x+1) will give incorrect results.

Syntax notation

For the notation of syntax of a MyM model the following conventions are used. Syntactic categories are indicated by *italic type*, and literal words and characters by `typewriter type`. Optional symbols are indicated by the subscript _{opt}.

Alternative categories are listed on separate lines. Enclosure by curly braces ({}) denotes zero or more repetitions.

2. Mathematical model

A mathematical model defined within MyM is a sequence of modules, global definitions, global declarations, and specifications.

```
model:
  { global-statement }
global-statement:
  module-definition
  constant-definition
  variable-declaration
  domain-specification
```

A *module-definition* is a definition of a submodel, in a *constant-definition* one or more constants are defined, with a *variable-declaration* the name, type, size, and default values of variables are declared, with a *domain-specification* attributes of domain variables can be set.

All such modules, constants and variables are globally known and may be referred to from within the modules.

The structure of a module is as follows:

```
module:
  module module-id ; begin { statement } end ;
statement:
  constant-definition
  variable-declaration
  domain-specification
  module-instantiation
  equation
```

The header of a module contains the identifier of the module, followed by a sequence of statements (local declarations, local definitions, instantiations, specifications, and equations), enclosed by begin and end. Each statement is terminated with a semi-colon. The order of statements is arbitrary, except that local variables and constants must be declared and defined before they are used.

The local *constant-definition* and *variable-declaration* statement are the same as the global versions, except that the defined constants and declared variables are only locally known. With an *module-instantiation* statement an instance of a module is created, with a *domain-specification* various attributes of domain variables can be specified, in an equation a dependent variable is equated to other variables.

Each MyM model should contain one module with the name `main`. In the following sections first the various statements are described. In section “Modules” on page 39 the use of modules will be discussed in more depth.

3. Constant definitions

With a constant definition one or more constants are symbolically defined.

constant-definition:

```
const  const-id = const-expr {const-id = const-expr};
```

Here *const-id* is a unique identifier, and *const-expr* a constant expression. The type of the constant (double, real, integer, or boolean) is the same as the type of the constant expression. The boolean constants `true` and `false` are predefined.

Also the integer constants `RK1`, `RK2`, `RK4`, and `RKV` are predefined. These are used for the selection of numerical solution methods. They denote the first, second, and fourth order Runge-Kutta method, and Runge-Kutta with a variable, adaptive time-step.

Examples:

```
const PI = 3.14159;  
const N = 24;  
const M = N*2;
```

A constant expression is the same as a standard expression, except that no variables may be used.

4. Variables

Variables are a central concept in MyM. They have many attributes, and come in several flavors. Each variable is identified with a unique name.

Type

A variable has an associated type: `real`, `double`, `integer`, `boolean`, and `void`. Real variables are single precision floating point numbers, double variables have double precision. Boolean variables have true and false as possible values. Void variables have no type, and can be used as place-holders before further refinement of the model. In section “Void functions” on page 37 their use is described. The name and type of variables follow from their declaration.

Examples:

```
real r;  
integer n;  
boolean alarm;
```

Size

Variables can be scalar variables or arrays. The latter category corresponds with subscripted variables, e.g. the MyM notation `m[i,j]` corresponds to the mathematical notation m_{ij} . Up to twenty subscripts are allowed. Subscripts range from 1 to a fixed upper limit. The number of subscripts and their upper limits follow from their declaration. Examples:

```
real r;  
real p[10,5,3];
```

where `r` is scalar and `p` is an array with 150 elements.

Use

A variable can be a standard model variable, an auxiliary variable, or a domain variable.

Most variables will be *model variables*. In several ways values can be assigned to these variables, which can be inspected with one of the user interfaces of MyM.

Auxiliary variables are used as index variables or to simplify righthand expressions (see “Loops” on page 22). Once a variable is used for the first time in the left side of a loop specification, it is recognized as an auxiliary variable. Auxiliary variables must be scalar. They can be used in other loop specifications, but may not be used as model variables. The user interfaces of MyM hide auxiliary variables: their value and their dependencies are not shown. The predefined variables *i*, *j*, and *k* are auxiliary variables.

Domain variables refer to the domains of functions. The most important domain variable is the predefined variable *t*. The time *t* is the domain of time-dependent variables. It is used for the numerical integration of the differential equations as free variable. Variables can be functions of other domains, for example height, temperature, or weight. If a variable is declared as such, the free variable is recognized as a domain variable. Domain variables should be scalar double precision variables.

A domain variable has several special attributes, such as its minimum and maximum, which are used as default specifications for the simulation and visualization. Examples:

```
double age;  
real r(age);  
real d[10];  
integer index;  
r(age) = max(100.0 - age, 0.0);  
d[index] = r(index)*2, index = 1 to 10;
```

Here x and d are model variables, $index$ is an auxiliary variable, and age is a domain variable. Time variables are a special class of domain variables. They are declared with the keyword `time`:

```
time my_t;  
real x(my_t);
```

Here the time variable `my_t` is declared, and x is declared as a function of `my_t`. The function and value of all time variables is the same as the standard time variable t .

However, the attributes `sample` and `step` can be specified independently. With this option the user can sample variables with different frequencies and use a different time step for parts of the models. This last option should be used with caution, however (see also section “Processing” on page 42).

Domain dependency

A model variable can have a constant value, or it can vary as a function of a domain variable. The values of variables in the first category are calculated initially at the start of a simulation. The values of variables in the second category are calculated from interpolation of tables, or are numerically evaluated during the simulation. The declaration of variables determines the category.

Examples:

```
real r;  
real f(t);
```

Here r is a variable with a constant value, and f is a function of t .

Variable dependency

A model variable can be an input variable or a dependent variable. If a variable occurs at the left side of an equation, it is recognized as a dependent variable. Its value is derived from the right side of the equation, and cannot be input by the user of a model.

All other model variables can be changed by the user. An initial default value must be specified as part of the declaration. Examples:

```
real r = 10.0;  
real s;  
s = 2.5*r + 4;
```

Here `r` is an input variable, and `s` is a dependent variable.

Scope

The term scope refers to the context where a name of a variable is known. The scope of a variable can be local, global, import or export.

Global variables are variables declared outside modules. Their names can be used in any module.

Local variables are variables declared inside modules. If the keywords `import` or `export` are used before a local variable declaration, then these variables can be used in other modules that instantiate this module. Examples:

```
real r = 10.0;  
module SCALE;  
begin  
  import real x;  
  export real y;  
  real f = 1.0;  
  y = f*x;  
end;
```

Here `r` is a global variable, `f` is a local variable, `x` is an import and `y` is an export variable. More on this in section “Modules” on page 39.

5. Variable declarations

In its simplest form, a declaration is a keyword that denotes the type, followed by a comma-separated list of variables. The type can be `real`, `double`, `integer`, `boolean`, `void`, or `time`. With this last type new time variables can be declared. For locally declared variables also the scope of a variable can be defined.

To declare an array, the sizes of the dimensions are appended after the variable, enclosed by square brackets. These sizes must be constant integer expressions. Up to twenty dimensions are allowed.

To declare a function, a domain variable enclosed by parentheses is appended. Suitable domain variables are real variables that are not used in the left side of equations, the predefined time variable `t`, or other time variables. Only one domain variable is allowed for functions.

```

declaration:
    scopeopt type var-declaration { , var-declaration } ;
    extern-func-declaration

scope:
    import
    export

type:
    real
    integer
    boolean
    time

var-declaration:
    var-id array-declopt domain-declopt value-declopt array-decl:
    const-expr { , const-expr } ]

domain-decl:
    ( domain-id )
    
```

The declaration of external functions is discussed in section “External functions” on page 34. The default value for input variables can be defined in the declaration. If it is omitted, all values are initialized to `o.o`, `o`, and `false`.

Two types of specifications can be used. To initialize variables with a constant value, an equals-sign followed by a comma-separated list of values can be used. For scalar variables, only one value must be used, for arrays the number of values should match the number of elements of the array. Multi-dimensional arrays are filled in row-major order: the last subscript runs fastest.

To initialize variables that depend on a domain variable with a varying set of values, an equals-sign followed by a list of values, enclosed by square brackets must be used. The values in this list are a series of blocks, consisting of the value of the domain-variable, followed by the value(s) of the variable itself. The number of blocks is not limited. The values of the domain-variable should be in increasing order.

For more flexibility, these initial values can also be read from an external file or from an external programme. The format is an equals-sign, followed by a `FILE` or `PROG` keyword, and a filename or a command (possibly with parameters), enclosed by parentheses and double quotes. The required data-format, and utilities for the preparation of such data is described in the MDIO api memo [MDIO].

```
value-decl:
  = var-values
  = [ domain-value, var-values { , domain-value, var-values } ]
  = FILE ("filename")
  = PROG ("command")
var-values:
  const-expr { , const-expr }
domain-value:
  const-expr
```


Examples:

```
real a, b[10](t), c = 4;
real d[2] = 10.0, 20.0;
real e[2,3](t) = [0.0,
                  0, 0, 0,
                  0, 0, 0,
                  8.0,
                  6, 5, 2,
                  4, 3, 2 ];
real x;
real f(x) = [ 0, 20, 1000, 15, 3000, 0 ];
real y = FILE("y.dat");
real z(t) = PROG("cat z.dat");
```

Here `a` and `c` are scalar variables; `b` is an array with ten timedependent elements; `d` is an array with two elements, which are initialized to 10.0 and 20.0; `e` is a time-dependent 2×3 array, which is initialized with varying values for time; `f` is a scalar variable, which depends on a domain variable `x`. Variable `y` is initialized with the contents of a file `y.dat`, and `z` is initialized by reading the output of the command `cat z.dat`.

6. Domain specifications

With a domain specification the initial values of various attributes for the simulation and visualization can be specified. All these attributes relate to domain-variables.

```
domain-specification:
    domain-id . domain-attribute = const-expr ;
domain-attribute:
    min
    max
    step
    sample
    relerror
    method
```

Domain-id is the name of a domain variable (e.g. t). The attributes `min` and `max` denote the lower and upper bound of the domain. For t they denote the start time and end time of the numerical simulation.

The attribute `step` denotes the step size used for numerical calculations, while `sample` denotes the sample rate for storing the results. The sample rate actually used is always rounded to an integral multiple of the numerical step size. The attribute `method` denotes the numerical method used for the solution. Currently this attribute has only effect for integration with respect to time variables.

Four methods are available: first, second, and fourth order Runge-Kutta, and Runge Kutta with a variable, adaptive time-step. A short description of these integration methods is given in section section “Differential equations” on page 29.

The predefined integer constants `RK1`, `RK2`, `RK4`, and `RKV` should be used for the specification of the method. The attribute `relerror` is used by the variable time-step method to specify the required accuracy.

Examples:

```
t.min = 0.0;
t.max = 1.0;
t.step = 0.1;
t.sample = 0.1;
t.relerror = 0.001;
t.method = RK2;
```

The values shown in this example are the defaults that are used if the user does not specify the values explicitly. The values specified in the model (explicitly or implicitly) are default values, and can be changed from the user interfaces. For time variables other than `t`, only `step` and `sample` can be specified independently. For the other attributes the values of `t` are used. For each attribute of a domain variable only one specification in a MyM model is allowed.

7. Equations

A standard equation is a specification that a dependent variable at the left side is equal to an expression at the right side.

It should not be confused with an assignment statement in a programming language. The assignment $x = x+1$ is a perfectly valid statement in a language such as FORTRAN or C, but it is not a sensible equation, since it does not hold for any x .

After the expression (see section “Expressions “ on page 20), optional loops can be specified (see section “Explicit loops “ on page 23).

Besides this standard format, two other type of equations can be used.

External procedures and subroutines that do not return a value can be called (see section “External functions” on page 34). Here a relation between the input and output parameters is established, instead of a relation between left side and right side variables.

With output-equations (see section “Output equations” on page 38) the result of a simulation can be written to a file, or passed to an external programme.

```
equation:
  dependent-var = expression loopsopt ;
  external-procedure-call loopsopt ;
  output-equation;
```

Dependent variables

A dependent variable at the left hand side of an equation starts with a variable identifier, followed by an optional variable subscript and an optional variable domain.

A variable identifier is simply the identifier of a variable, or the identifier of an instantiated module, followed by a full-stop and the identifier of a variable. This last option will be discussed further in section “Modules” (see section “Modules” on page 39).

The variable subscript is a comma- separated list of integer expressions, enclosed by square brackets. The variable subscript can be used only for arrays, and the number of expressions should match with the number of dimensions of the variable. However, the variable subscript is not obliged for arrays (see section “Implicit loops” on page 22). If a subscript is used, the equation holds only for a single element of the array.

Further, the variable domain (if present) can be appended: the name of the domain variable, enclosed by parentheses. This option is introduced for notational convenience, for the processing of

the model it does not matter whether the domain is appended or not.

```
dependent-var:
  variable var-subscriptopt var-domainopt
variable:
  var-id
  instance-id . var-id
var-subscript:
  [ expression { , expression } ]
var-domain:
  ( domain-id )
```

Conversion

If the dependent variable and the expression at the right side have the same type, no problems arise. If the dependent variable is a real or double variable and the expression is of integer type, the result of the expression is converted to real or double. This might introduce a round-off error. If the dependent variable is an integer variable and the expression is of real or double type, the result of the expression is truncated. Boolean dependent variables can only be used in combination with boolean expressions.

8. Expressions

The general form of an expression is a primary expression, possibly preceded by a unary operator (– or not), or two expressions to which a binary operator is applied.

```
expression:
  primary-expression
  – primary-expression
  not primary-expression
  expression operator expression
```

The following binary operators are available, with decreasing priority as indicated.

```
operator:
  **
  *      /
  +      –      >      <      >=      <=      =      <>
  and    or     not
```

The ** operator is the exponentiation operator. Integer expressions used in combination with the exponentiation operator are converted to real before the exponentiation is done. If a binary operator is applied to a real (or double) and an integer expression, then the result is a real (or double) expression.

If an expression is divided by an integer expression, the result of the integer expression is always converted to double before the division. The result of a division is therefore always double.

The operators not, and, and or can only be applied to boolean expressions, the other operators only to numerical expressions.

Primary expression

The simplest primary expressions are a constant identifier, a constant (double, real, or integer), or a possibly subscripted variable identifier. Parentheses can be used to group expressions. Also a domain identifier can be used as a primary expression, but here restrictions apply.

The general rule is that the dependent variable should be a function of the same domain as used in the expression. For more information see section “User functions” on page 26. Also the value of domain attributes can be used in expressions.

```
primary-expression:  
  const-id  
  constant  
  variable var-subscriptopt  
  ( expression )  
  domain-id  
  domain-id . domain-attribute  
  function-call
```

Some examples of equations with various expressions:

```
const A = 4.0;  
real thalf, x[2](t), y(t);  
  
thalf = (t.min + t.max)/2.0;  
x[1] = A*(t - thalf);  
x[2] = A*A*(t - thalf)**3;  
y(t) = (x[1] + x[2]);
```

Many options are available for functions. They are discussed in section “Functions” on page 25.

9. Loops

The MyM language provides several constructs for loops to simplify the manipulation of arrays. Implicit and explicit loops are treated in this section, loop functions are dealt with in section “Loop functions” on page 32.

Implicit loops

It often occurs that in a model several arrays with the same number of dimensions and sizes are used, where the relation between the elements is the same for all elements.

To handle this situation, the following rule is used. If the dependent variable has one or more dimensions, and if no subscript is used, then in the expression at the right side of an equation both scalar variables and arrays with the same dimensions and sizes as the dependent variable can be used. The expression is then applied to all elements of the arrays in the same way.

Example:

```
real a[2, 3] = [1, 2, 3, 4, 5, 6];  
real b[2, 3];  
b = a*a + 3*a + 4;
```


Explicit loops

For more control, one or more explicit loops can be appended to expressions. The mixed use of explicit and implicit loops is not allowed.

```
loops:
  , simple-loop { , simple-loop }
simple-loop:
  variable = start-value end-valueopt step-valueopt
start-value:
  expression
end-value:
  to expression
  downto expression
step-value:
  step expression
```

The first element of a loop is a variable identifier. This loop variable is recognized as an auxiliary variable, and is further hidden from the user interfaces. It may not be used at the left side of an equation. It can be an integer or a real variable. Note that real variables are not exact, and rounding errors can lead to unexpected results.

Next, the start value, the end value, and the step size of the loop variable are specified, separated by the keywords `to` or `downto`, and `step`.

If `to` is used, then the loop variable is incremented with the step size until its value is greater than the end value.

If `downto` is used, the loop variable is decremented with the step size, until its value is lower than the end value. The step size is optional, and by default assumed to be equal to 1. Also, the end value can be omitted.

In this case, the loop is only executed once, using the start value for the loop variable. If several loops are used, then the loop variable of the last loop runs fastest. In other words, each loop is nested within its preceding loop, if present. Therefore it is not allowed to refer in the expressions of a loop to loop variables of succeeding loops.

Examples:

```
real r = 10.0, a, x[4], y[4,4];  
x[1]= a*a + a + 1, a = r**2.5 + r**1.5;  
x[i]= 0.8*x[i-1], i = 2 to 4;  
y[i,j] = 2*x[i], i = 1 to 4, j = 1 to i;  
y[i,j] = 3*x[i], i = 1 to 4, j = i+1 to 4;
```

Here `a` is used as an auxiliary variable to simplify an expression. The last two examples show the use of multiple loops. If the start value exceeds the end value, the loop is not executed at all. Make sure that all elements of dependent arrays are used in the left hand side of equations, else the results are undefined.

10. Functions

The MyM language provides the user with a large set of standard functions, as well as the possibility to define his own functions. The general format of a function call is the name of the function, followed by a comma-separated list of expressions, enclosed by parentheses.

function-call:

function-id (expression { , expression }

As will be discussed later, for some functions special restrictions and extensions apply. If a function expects a real expression and a integer expression is used, then this integer expression is converted to real automatically.

Mathematical functions

A standard set of mathematical functions is available. All functions have a double precision value. The following standard functions can be used. Trigonometric functions use radian arguments or return a value in radians.

<code>sin(x)</code>	sine
<code>asin(x)</code>	arc sine
<code>sinh(x)</code>	hyperbolic sine
<code>cos(x)</code>	cosine
<code>acos(x)</code>	arc cosine
<code>cosh(x)</code>	hyperbolic cosine
<code>tan(x)</code>	tangent
<code>atan(x)</code>	arc tangent
<code>atan2(y, x)</code>	arc tangent of y/x
<code>tanh(x)</code>	hyperbolic tangent
<code>exp(x)</code>	exponential function e^x
<code>log(x)</code>	natural logarithm
<code>pow(x, y)</code>	x^y
<code>sqrt(x)</code>	square root \sqrt{x}
<code>fabs(x)</code>	absolute value
<code>fmod(x, y)</code>	x modulo y
<code>floor(x)</code>	$\lfloor x \rfloor$
<code>ceil(x)</code>	$\lceil x \rceil$

The preceding functions are the same as the standard functions of the C programming language. For a more detailed description, consult the C manual. For convenience, the functions `abs(x)` and `mod(x, y)` have been added, which have the same effect as `fabs` and `fmod` respectively.

Two functions are available that produce random numbers:

<code>uniform(a,b)</code>	random number from the uniform distribution over the interval $[a,b]$
<code>gauss(m,s)</code>	random number from normal distribution with mean value m and standard deviation s

Three functions are available with a argument list of variable length:

<code>min(x,y,...)</code>	minimum value of arguments
<code>max(x,y,...)</code>	maximum value of arguments
<code>avg(x,y,...)</code>	average value of arguments

User functions

As mentioned before, variables in MyM can depend on a domain or free variable. Such variables are in effect functions. In this section we consider such variables or user defined functions. First variables that depend on t are considered, next variables that depend on other domain variables.

The value $x(t)$ can be set in two different ways. First, if a variable is an input variable, a default value (constant or time-dependent) can be set with the declaration. This default value can later be overruled via the user interfaces. Second, if a variable is a dependent variable (i.e. occurs at the left side of an equation) it is calculated, via integration or via an algebraic equation.

Within a MyM model the notion of a current time t is used. The value of t progresses from t_{\min} to t_{\max} as the simulation proceeds. The value of t may be used in expressions. Time dependent variables always refer to their value at the current time t . Thus, the following statements are all equivalent:

```
y(t) = 2*x(t) + t;
y(t) = 2*x + t;
y = 2*x(t) + t;
y = 2*x + t;
```

In the right-hand side of an equation, also a real expression may be used as an argument for time dependent variables. For such variables, with an expression not equal to `t` used as argument, the system stores all calculated values at resolution `t.step`, and interpolates these data to determine the requested value.

For calculated variables only data from `t.min` to `t` are available. If the value of the free variable is lower/higher than the values stored, the first/last value of the function is returned. In many cases the full generality of this interpolation mechanism is not required, and only the last value of a variable is of interest. To this end the following function is provided:

```
last(var, default)
```

where `var` is a possibly subscripted time dependent variable. Its value is the value of `var` at the last (complete) time step. The initial value is default. Examples of some difference equations which can be defined with these options:

```
real x(t), y(t), delay = 1.0, z[5](t);
y(t) = switch(t-delay > t.min ? x(t-delay)
              else 0.0);
z[1](t) = 1.0;
z[i](t) = 0.5*last(z[i-1], 0.0), i = 2 to 5;
```

The `switch` function is discussed in section “Conditional functions” on page 32 .

With `last` only the value for the last complete time step can be retrieved. An alternative is the function:

```
nlast(var, n, default)
```

here `var` is again a possibly subscripted time dependent variable, `n` is the number of complete time steps to go back, and `default` is the value to be used if this value can not be determined. If `n=1` then the value of `nlast` is the same as `last`. The difference between `nlast` and the use of an expression between parentheses after a time-dependent variable is that for `nlast` no interpolation of the stored values is used (the value at a computed step is used), while with the other method stored values are interpolated.

For functions of user defined domain variables the following rules apply. First, default values can be defined via the declaration. Second, the value can be defined via an equation. Here it is obliged that in the right side of the equation only one domain variable is used, namely that of the variable.

The use of `t` or of functions of `t` is not allowed. With respect to the use of such functions in expressions, always a value for the domain variable should be specified. In contrast to `t`, no current value for user defined domain variables is available that can be used as a default.

Finally, the use of arrays of functions of user defined domain variables is not allowed. Examples:

```
real x, f(x), g(x) = [0, 10, 100, 0];  
f(x) = sqrt(x) + x*x;
```

Differential equations

MyM can handle systems of differential equations. The `integ` function must be used to define them. A differential equation can have two forms:

```
differential-equation:
  dependent-var= integ( derivative, initial ) loopsopt;
  dependent-var= integ( derivative , initial , condition ) loopsopt;
```

Here *derivative* is a real expression that denotes the derivative of the dependent variable and *initial-value* a real expression that denotes its initial value.

The `integ` function cannot be used combined with other expressions. If used, it completely occupies the expression at the right hand side of an

equation. Loops, implicit or explicit, may be used however.

As an example, the following MyM statement:

```
real r = 5.0, y(t);
y = integ(y*y + y + 3, r);
```

is equivalent to the following definition of $y(t)$ as an integral:

$$y(t) = r + \int_{t_{min}}^t y^2 + y + 3 \, dt'$$

Also, it is equivalent to the following initial value specification and differential equation:

$$\begin{aligned} y(t_{min}) &= r \\ y' &= y^2 + y + 3 \end{aligned}$$

The standard form of `integ` takes two parameters. The output is initialized at $t = t_{min}$ to the initial value, and in the following time the derivative is integrated. With the three parameter form of `integ` the integration can be reset. At $t = t_{min}$ and further at each time the condition holds true, the initial value is assigned to the integrated variable. Both the initial value and the condition can be expressions, as long as no circular static dependencies are introduced. As an example, if the variable `y` must be reset at all multiples of $t = 10$, then the following equation can be used:

```
y = integ(f, 0.0, mod(t, 10.0) < 0.001);
```

All integrations are done with time variables as free variables, starting with $t = t_{min}$. The method to be used must be specified with the method attribute of `t`. These methods are implemented according to the description in [Recipes]. A short overview. Consider a single variable $y(t)$ to be integrated, with derivative $y'(t, y)$. Its value at step n is denoted by y_n . The simplest integration method (RK1) is the first order Runge-Kutta or Euler method. In each time step y is incremented according to:

$$y_{n+1} \leftarrow y_n + y'(t_n, y_n) \Delta t$$

where Δt is equal to `tstep`, and

$$t_n = t_{min} + n \Delta t.$$

For the second order Runge-Kutta method (RK2) the following scheme is used:

$$\begin{aligned} k_1 &\leftarrow \gamma'(t_n, \gamma_n) \Delta t \\ k_2 &\leftarrow \gamma'(t_n + \Delta t / 2, \gamma_n + k_1 / 2) \Delta t \\ \gamma_{n+1} &\leftarrow \gamma_n + k_2. \end{aligned}$$

Here a trial step is used to the midpoint of the interval. For the fourth order Runge-Kutta method (RK4) the scheme is:

$$\begin{aligned} k_1 &\leftarrow \gamma'(t_n, \gamma_n) \Delta t \\ k_2 &\leftarrow \gamma'(t_n + \Delta t / 2, \gamma_n + k_1 / 2) \Delta t \\ k_3 &\leftarrow \gamma'(t_n + \Delta t / 2, \gamma_n + k_2 / 2) \Delta t \\ k_4 &\leftarrow \gamma'(t_n + \Delta t, \gamma_n + k_3) \Delta t \\ \gamma_{n+1} &\leftarrow \gamma_n + k_1 / 6 + k_2 / 3 + k_3 / 3 + k_4 / 6 \end{aligned}$$

The derivative is evaluated four times here: once at the initial point, twice at trial midpoints, and once at a trial endpoint. From these derivatives the new value is calculated.

With all these methods it is the responsibility of the user to specify a suitable time step. With the Runge-Kutta method with variable, adaptive time step (RKV) the user can specify a desired accuracy, and the time step used is decreased, if necessary, to meet this specification. The principle of the method is as follows. First, a single fourth order Runge-Kutta step is made, which gives a result γ_1 . Second, starting from the same point, two fourth order Runge-Kutta steps are made with half the time step, giving γ_2 . The results are compared, and an estimate of the error is made. If this error exceeds the allowable error (`relerror`), then a new estimate for the time step is made and the procedure is repeated, else the integration proceeds.

This process is used for the intervals with width t_{step} . If a variable time step exceeds the boundary of such an interval, the variable time step is truncated. Intermediate results are not stored for interpolation purposes, this is only done for all t_n 's.

A good general purpose choice for the error is the relative error. However, if a variable is equal to or close to zero, the use of a relative error can lead to division by zero. Therefore the following definition (see [recipes]) is used:

$$\text{error} = |y_2 - y_1| / (|y_n| + |y'(t_n, y_n)| + \varepsilon)$$

with $\varepsilon = 10^{-20}$.

Loop functions

Loop functions are provided to simplify operations on arrays. The general format is:

loop-function:
function-id (simple-loop , expression)

The first argument is a loop, the second an expression to which this loop is applied. The function name determines the result. The following loop functions are available:

<code>lsum(l,e)</code>	sum of e
<code>lprod(l,e)</code>	product of e
<code>lavg(l,e)</code>	average of e
<code>lmin(l,e)</code>	minimum of e
<code>lmax(l,e)</code>	maximum of e

As an example, the following MyM equations:

```
p = lsum(i = 1 to N, v[i]*v[i]);
s = lsum(i = 1 to N, lsum(j = 1 to N,
    a[i,j]));
```

correspond to the following mathematical equations:

$$p = \sum_{i=1}^N v_i v_i$$

$$s = \sum_{i=1}^N \sum_{j=1}^N a_{ij}$$

Conditional functions

Conditions can be handled with the switch function.

The syntax is :

```
switch-expr:
    switch ( cond-expr { , cond-expr } else default-value)
cond-expr:
    condition? expression
```

Here condition is a Boolean expression. If true, the value of the expression behind the question mark is used. If false, the next condition is evaluated. If all conditions fail, the default value (again an expression) is used. As an example, the following MyM statement:

```
real a(t);
a(t) = switch (t < 0 ? 0,
    t < 10 ? t
    else 10);
```

is equivalent to the following equation :

$$a(t) = \begin{cases} 0 & \text{if } (t < 0) \\ t & \text{if } (t \geq 0) \text{ and } (t \leq 10) \\ 10 & \text{otherwise} \end{cases}$$

External functions

If the mathematical functions offered fall short, existing functions coded in C or Fortran can be called. They must be declared in the MyM-model, so that the type of the function and its parameters are known. The syntax of the declaration is:

```
extern-func-declaration:
  extern languageopt (external-type | void )
    function-id parameter-list ;
language:
  C | FORTRAN
external-type:
  float | real | int | integer | double | logical
parameter-list:
  ( parameter { , parameter } )
parameter:
  exportopt external-type | external-type *
```

This declaration must match exactly with the implementation of the external function. The mixed use of `float` and `double` will lead to errors. Two types of parameters can be used.

For the transfer of the value of scalar parameters, specify only the type of the parameter. As actual parameter all expressions can be used, as long as the types of the actual expression and the declared parameter can be matched. Integer expressions are automatically converted to real or double if required.

For the transfer of the address of parameters, specify the type of the parameter, followed by an asterisk (*). This is useful for passing arrays. As actual parameter for arrays the name of a variable (possibly prefixed with a module name) must be used. Loop-expansion is not used here.

By default, the assumption is made that the external function does not modify the contents of parameters that are passed by address. If this assumption is violated, the results of simulations will be erroneous.

However, if the keyword `export` is put in front of the type, then it is assumed that this variable is an output-variable. In other words, this variable is assumed to be dependent on all other inputvariables. Also, the variable to which the value of the function is assigned is assumed to be dependent on all input-variables that are used in the parameter list of the external function called.

A simple example. The following FORTRAN function takes an array `X` with `N` elements, and returns the sum, scaled with `R`.

```
REAL FUNCTION SUMSCA(N, X, R)
  INTEGER N
  REAL X(N), R, S
  S = 0.0
  DO 10 I = 1, N
    S = S + X(i)
  10 CONTINUE
  SUMSCA = R*S
  RETURN
END
```

Obviously, this can also (and even simpler) be expressed with MyM equations, but this is just an example. This function can be used by a MyM model as follows:

```
extern FORTRAN real sumscs(int,real*,real);
module main
  begin real a[4](t), b(t);
    a[i] = t**i, i = 1 to 4;
    b = sumscs(4, a, 2);
  end;
```

Finally the Fortran function must be compiled, and the resulting object-file must be linked with the object-file that is the result of the processing of the MyM model. Another example, where the resulting values are passed via a parameter.

```

SUBROUTINE MVMUL(M, N, A, V, W)
C
C Calculate the matrix-vector
C product W=A*V
C
      INTEGER M, N, I, J
      DOUBLE PRECISION A(N,M), V(N), W(N), R
C
      DO I = 1, M
        R = 0.0
        DO J = 1, N
          R = R + A(J,I)*V(J)
        END DO
        W(I) = R
      END DO
C
      RETURN
END

```

This subroutine can be invoked from a MyM model as follows.

```

extern FORTRAN void mvmul(integer, integer,
double *, double *, export double*);
const M = 3, N = 2;
double Av[N], v[N] = 1, 2;
double A[M,N] = 1, 2, 3, 4, 5, 6;
module main
begin
    mvmul(M, N, A, v, Av);
end;

```

Note that for such a subroutine the return value does not have to be used. Such a procedure-call is considered as an equation by MyM. Further, the ordering of the arrays is different in Fortran and in MyM. Finally, with this mechanism it is not possible to use a variable both for input and output.

Void functions

In an early phase of the modelling of a complex phenomenon, only for a part of all relations between variables strict mathematical equations will be available to describe these relations. The MyM language provides variables of the type void and the `dependent()` function to describe such relations.

A void variable is declared in the same way as all other variables. Dimensions and domains can be used. Its use in equations however is restricted. The use of void variables and void functions in the right side of equations is only allowed if the left side (the destination) is also of type void. The `dependent()` function has been defined to make the use of such informal relations more easy and explicit. The number of parameters of this function is variable, and can these can be of any type. The result of the function is void.

An example:

```
void x1, x2, x, y1, y2, y, z;  
module  
main begin  
    x = dependent(x1, x2);  
    y = dependent(y1, y2);  
    z = dependent(x, y);  
end;
```

Here the dependencies between six variables are described: a binary tree. This MyM model can be processed and the structure of the model can be visualized.

11. Output equations

In section “Variable declarations” on page 13 was described how variables can be initialized with external data from a file or from a programme. A similar format can also be used for the output of data: the output-equation.

```
output-equation:
  output-spec = variable-id ;
output-spec:
  FILE ("filename" output-formatopt)
  PROG ("command" output-formatopt)
output-format:
  , "a"
  , "b"
```

The variable specified left of the equals-sign is written to a file or to a programme. With the optional output-format a choice can be made between a ASCII or a binary format. The default for files is ASCII, the default for programmes is binary. Some examples:

```
real x(t), y(t);
module main
begin
  x = t; y = t*t;
  FILE("x.dat") = x;
  PROG("cat > y.dat", "a") = y;
end;
```

When all values of x and y are calculated, the values of x and y are written to files `x.dat` and `y.dat`. For a description of the data-format see [MDIO].

12. Modules

In the preceding sections a number of references were made to various aspects of modules. In this section an extensive description will be given of the module concept in MyM.

It is desirable that a large model, consisting of a large number of equations, can be split in manageable submodels, which are developed independently. Further, such submodels can be reused, both within the same model as well as shared between several models. The module concept provides a flexible solution for this.

An obvious solution for modules would be to mimic functions and subroutines provided by procedural languages such as C and FORTRAN. Here a subroutine is declared with a parameter-list with a fixed number of formal parameters. The subroutine is invoked by calling its name, followed by a list of actual parameters.

However, this solution has two important limitations. First, the fixed parameter-list is rigid. It forces the user of a module to specify all parameters. It would be more convenient if he could only specify the parameters that are special for his model, and rely on suitable defaults for other parameters. Second, it is impossible to distinguish between different calls of the same module. This is important for the graphical inspection of the data with the Graphical User Interface of MyM.

Therefore, a different solution was chosen. The principle is that by defining a module a new type or class is defined. As an example, suppose we want to have a special module, called `mix`, that takes three values `x`, `y` and `a` as input, and returns a weighted value $z=(1-a)x+ay$.

We can define a module as follows:

```
module mix;
begin
  import real x(t)=0.0, y(t)=0.0,
           a = 0.5;
  real wx(t), wy(t);
  export real z(t);
  wx = (1-a)*x;
  wy = a*y;
  z = wx + wy;
end
```

To use this module we first must make an instance of it. The instantiation statement is defined as:

```
instantiation:
  module-id instance-id { , instance-id };
```

As an example, with

```
mix Amix, Bmix;
```

we define two instances of the `mix` module. Next we must connect the variables of these instances with the variables of the module in which they are instantiated. The variables `x(t)`, `y(t)`, and `a` are declared as import variables of the `mix` module. Their value can be imported from outside the module.

Note that default values are specified: if no value is specified, this default is used. This simplifies the definition of general purpose modules. Users of such modules can decide which input variables they want to set explicitly. Existing models do not have to be changed if the modules used become more complex, and have more import and export variables for more sophisticated usage.

The variables `wx(t)` and `wy(t)` are local variables. Their value is not accessible from outside the module. Finally, the variable `z(t)` is an export variable: its value can be exported outside the module. A module does not have a separate header in which the interface is defined. If the keywords `import` or `export` are used in a declaration, then these variables can be accessed from outside the module. The following example shows how:

```
real u(t), v(t), w(t);

Amix.x = u;
Amix.y = v;
Bmix.x = Amix.z;
Bmix.y = Amix.z;
w = Bmix.z;
```

Thus, the value of an import or export variable is referred to as the *instance-id*, a full-stop, and the *variable-id*. Import variables can only be used in the left side of an equation, export variables only in the right side of an equation.

Note that in this example output of the `Amix` instance is directly used as input in the `Bmix` instance.

Besides local, export, and import variables also global variables can be used: variables that are declared outside modules can be used in all modules.

One module has a special meaning: the module with the name `main`. This module is considered as the heart of the total model. Only modules that are instantiated (directly or indirectly) in this `main` module are included in the model.

13. Processing

All equations are checked by the MyM environment. Dependencies between variables are derived automatically. Two kinds of dependencies are distinguished.

In case of *static* dependency the dependent variable is dependent on a variable in the right side of the equation for the same value(s) of t .

In case of *dynamic* dependency the dependent variable depends on a variable for different values of t . This occurs if a variable is used as the first argument of `integ`, or if interpolation of old values is used.

Static dependency of a variable with itself, directly or via a roundabout, is not allowed. The following equations lead to error messages:

```
r = 3*r + 4;  
y = integ(x, y);
```

An exception to the previous rule are array-variables. The following equations are allowed, but result in a warning:

```
y[1] = 3.0;  
y[2] = y[1];
```

It is the responsibility of the user to make sure that in such equations the left- and right variables are different, and that the equations are written down in the right order. The following equations can be handled without problems:

```
r = 3*r(t-1) + 4;  
y[1] = integ(y[2], 0.0);
```

After the check, all equations are sorted in a correct order, to make sure that values are calculated before they are used.

The values of time dependent variables are evaluated if the time domain variable has a value of t_{min} plus an integral number of times the step size. This can give surprising results if multiple time domain variables are used with different step sizes. In the following example for instance, the value of $p(t)$ is not 0 !

```
real p(t);
time my_t;
t.step = 1;
my_t.step = 10;
p = t - my_t;
```

Here my_t is only set at each 10 steps of t , hence in between their value is constant. This is a simple situation, but in more complex ones such effects can be quite subtle. The use of this option is therefore discouraged for non-experienced users.

Another restriction concerns the values used for the step size and sample rates. Before the evaluation of the model the smallest step size of all time variables is determined, and all other step sizes and sample rates are rounded to integral multiples of this value.

14. References

- [FTV] W.H., Flannery, B.P, Teukolsky, S.A., Vetterling, W.T.,
Numerical Recipes in C : The Art of Scientific Computing,
Cambridge University Press, Cambridge, 1988.
- [MDIO] www.my-m.eu, *MDIO Application Programmers Interface*, 2008.
- [Recipes] *Numerical Recipes: The Art of Scientific Computing*, 2007.
- [Language Tutorial] Tizio/Netherlands Environmental Assessment Agency, *MyM Language Tutorial*, 2008.

