Documentation of model building in Mobius

Magnus Dahler Norling

August 6, 2019

Contents

1 Introduction

This is documentation for model developers. Documentation for model users and framework developers will be provided in separate documents (eventually).

Note 1. It is recommended for any model developers to take a look at the tutorials and experiment with making changes to them before reading all of the documentation. The tutorials will also provide the information about how to compile the models into finished exes. After you have some experience, it is also a good idea to look at existing models (in the Modules folder of the github repository) for more advanced examples.

2 Basic concepts

2.1 The model

The model object contains all immutable information about the model that will not change with each particular usage of the model. The model object contains lists of various model entities and information about these that are provided by the model developer in the model registration routine(s). The different model entities are presented in Table 1.

Index sets	Parameters, inputs and equation results can index over one or more index sets. For instance, you may want to evalute the same equations in multiple contexts, such as once per reach in a river, and in that case you want an index set for these reaches.
Parameter groups	Parameter groups are collections of parameters. A parameter group can vary with an index set, making each parameter have a separate value for each index in the index set. The parameter group can also be a child group of another parameter group. In that case, each parameter in the group has a separate value for each pair of indexes from the index set of its group and the parent group. This can be chained to make parameters vary with as many index sets as one wants.
Parameters	Parameters are values used to tune the equations of the modell. Parameters do not vary over time. Currently four different types of parameters are supported: double precision floating point, unsigned integer, boolean and time.
Inputs	Inputs are forcings on the model that vary over time. An example of an input is a time series of daily air temperature that has been measured in the field. Inputs can also vary with index sets. For instance, one can have a separate air temperature series for each subcatchment area in the model.
Equations	The model has a set of equations that are executed in a specific order for every timestep of the model (see later descriptions of how the run order is determined). Each equation can look up the value of various parameters, inputs, results of other equations (both from the current timestep and earlier) and produces a single output value every time it is evaluated. There are several types of equations. The main two are discrete timestep (or basic) equations and ordinary differential equations (ODE).
Solvers	Each solver contains a list ODE equations that it will solve over one timestep whenever it is run. The solver must be registered with additional information about e.g. which integrator method it will use.

Table 1: The model entities

2.2 The dataset

The dataset contains information about a specific setup of a model. This includes all the indexes of each index set, as well as values for all the parameters and input series in the model. It will also contain the result series of each equation after the model is run. The typical way of setting up a dataset is by reading it in from a parameter file and an input file. The file formats for these are documented separately.

3 Model structure

3.1 Index sets and indexes

Index sets is one of the fundamental concepts in the Mobius framework. Examples of index sets can for instance be "Reaches", or "Landscape units". Parameters, equation results and inputs can index over index sets, so that the same equation is evaluated for many indexes, and with different parameter values for each (combination of) indexes.

Example 1. The equation "Snow fall" depends on the input timeseries "Actual precipitation" and "Air temperature", and on the parameter "Canopy interception". Say that "Canopy interception" indexes over the index set "Landscape units". In this case, "Snow fall" will be evaluated once per index in the "Landscape units" index set, using the corresponding parameter value of "Canopy interception" for each index. If one of the inputs also index over "Reaches", then "Snow fall" will be evaluated once per reach-landscape unit pair. This generalizes to higher number of index sets.

Which index sets each model has is fixed, and determined by the model developer. However, which indexes each index set contains can usually be determined by the model user, for instance by configuring a parameter file. Some models do however require a fixed set of particular indexes for some of its index sets. For instance INCA-N only works with three particular soil boxes.

There are two types of index sets

i Basic index sets. These have no additional structure except for the order of the indexes.

ii Branched index sets. These contain additional connectivity information, and are typically used to encode river structures. Each index in a branched index set has a list of incoming branches, the branches being other indexes in the same index set.

3.2 Parameter structure

Each parameter belongs to a parameter group, which again can belong to another parameter group and so on. Moreover, each parameter group can index over an index set. This means that each parameter can in practice be viewed as a multidimensional array of parameter values, where each dimension in the array is indexed by one of the model's index sets. It is possible for a parameter to index over the same index set multiple times. For instance, the PERSiST hydrology model [1] has a "Percolation matrix" parameter where both the row and column in the matrix is indexed by the soil box.

Example 2. A PERSiST "Percolation matrix", from a parameter file. (See separate documentation for a detailed description of the parameter file format).

```
"Percolation matrix" :
0.2 0.8 0.0
1.0 0.5 0.5
0.0 0.0 1.0

0.1 0.9 0.0
1.0 0.6 0.4
0.0 0.0 1.0
```

The parameter indexes over "Landscape units" once (signifying what matrix you are looking at) and "Soils" twice (signifying which row and column). The percolation matrix is used to say something about what proportion of the water will flow between each soil box, but we will not go into detail about that here.

The parameter group and index set structure of the parameters are entirely fixed by the model (developer), and can not be configured in the parameter file. But one can usually configure which indexes each index set has and which values the parameter has for each (tuple of) index(es).

3.3 Input structure

An input is a timeseries (usually of measured data or preprocessed data that is based on measured data). Inputs are used as forcings in many models. For instance, hydrological models often use precipitation timeseries to determine how much water enters the system at each day. Inputs can either be global for the system, or like parameters can index over one or more index sets. For instance, one may want to have a separate precipitation timeseries for each subcatchment (reach). Unlike with parameters, which index set each input indexes over is determined in the input file. This can sometimes change the equation structure of the model. For instance, if precipitation and air temperature does not depend on which subcatchment one are in, the framework may decide for some models that equations like "Precipitation falling as snow" should only be evaluated once globally instead of per subcatchment. This does however also depend on which parameters these equations reference. If "Precipitation falling as snow" references a parameter that indexes over the "Reaches" index set, that equation will still evaluate once per reach.

3.4 Equation batch structure

During the main run of the model, the model will for each timestep evaluate all equations (sometimes called state variables) in a given order. Equations can also be evaluated for multiple indexes. For instance any equations having to do with flow or volume of a reach are typically evaluated for each index in a "Reaches" index set. Equations are sorted so that an equation is always evaluated after other equations that it uses the result values of. To facilitate this, the model builds an equation batch structure.

Example 3. The equation batch structure for the framework implementation of PERSiST.

```
**** Result Structure ****
[Reaches][Landscape units]
       -----
       Snow melt
       Rainfall
       -----

[Reaches][Landscape units][Soils]
       -----
       Percolation input
       Saturation excess input
       Input
       Water depth 1
       Saturation excess
       Water depth 2
       Evapotranspiration X3
       Evapotranspiration X4
       Evapotranspiration
       Water depth 3
       Total runoff
       Drought runoff
       Percolation out
       Water depth 4
       Runoff
       Runoff to reach
       Water depth
       -----

[Reaches][Landscape units]
       -----
       Snow fall
       Snow as water equivalent
       (Cumulative) Total runoff to reach
       Diffuse flow output
       -----

[Reaches]
       -----
       (Cumulative) Total diffuse flow output
       Reach flow input
       ----- (SOLVER: Reach solver)
       Reach time constant
       (ODE) Reach flow
       (ODE) Reach volume
       -----
       Reach velocity
       Reach depth
       -----
```

The model has an ordered list of batch groups, where each batch group has an ordered list of batches. Each batch group also has an ordered list of index sets that it indexes over. Each batch can either be a discrete batch or a solver batch. The model will evaluate each batch group at a time. Then for each combination of indexes in the index sets it evaluates each batch in the batch group in order. In discrete batches, each equation is evaluated in order. In solver batches, the ODE system will be integrated possibly evaluating each equation many times. In Example

3, the first three batch groups contain one batch each, while the last batch group contains three batches.

Note 2. The only time a batch group can have more than one batch is if it contains both a solver batch and one or more discrete batches (or possibly another solver batch using a different integrator). Two discrete batches next to each other that index over the same index sets will always be merged.

Example 4. Pseudocode for how the model evaluates the batch structure in Example 3.

```
for every timestep:
       for every reach R:
              for every landscape unit LU:
                     evaluate "Snow melt" and "Rainfall" for this LU in this R.
       for every reach R:
              for every landscape unit LU:
                     for every soil box S:
                            evaluate "Percolation input" to "Water depth" for this S
                                 in this LU in this R
       for every reach R:
              for every landscape unit LU:
                     evaluate "Snow fall" to "Diffuse flow output" for this LU in
                         this R
       for every reach R:
              evaluate "Total diffuse flow output" and "Reach flow input"
              run a solver that integrates the ODE system containing the equations "
                  Reach time constant" to "Reach volume" (possibly evaluating them
                  many times)
              evaluate "Reach velocity" and "Reach depth"
```

At the end of a model run, one can extract a timeseries of each of these equations for any combination of indexes that they index over. For instance one can look at the timeseries for "Snow melt" in {"Langtjern", "Forest"}, or the timeseries for "Water depth" in {"Langtjern", "Peatland", "Organic soil layer"} (given that these were indexes provided for this particular dataset).

Unlike parameters, equations can never index over one index set more than once. We will go through how the index set dependencies of an equation are determined and how it is placed in a batch structure later.

4 Model building

4.1 Setting up a project and creating a model object

We refer to the quick start guide on the front page of the github repository as well as the tutorials on how to quickly get a project started.

Usually you will have one header (.h) file with the procedures you use to build your model, and one source file (.cpp) with the main function of the C++ program, where you do all application-level operations such as creating and destroying model and dataset objects or reading in parameter values from files, running the model and so on.

Example 5. Example of a main.cpp

```
#define MOBIUS_PRINT_TIMING_INFO 1

#include "../mobius.h" //NOTE: This path is relative to the location of this file
#include "mymodel.h"

int main()
{
	const char *InputFile = "mytestinputs.dat";
	const char *ParFile  = "mytestparameters.dat";

	mobius_model *Model = BeginModelDefinition();

	AddTestModel(Model);

	ReadInputDependenciesFromFile(Model, InputFile);

	EndModelDefinition(Model);

	PrintResultStrucuture(Model);

	mobius_data_set *DataSet = GenerateDataSet(Model);

	ReadParametersFromFile(DataSet, ParFile);
	ReadInputsFromFile(DataSet, InputFile);

	RunModel(DataSet);

	u64 Timesteps = GetTimesteps(DataSet);
	PrintResultSeries(DataSet, "AX", {}, Timesteps);
}
```

Example 6. mymodel.h

```
static void
AddTestModel(mobius_model *Model)
{
	auto Days = RegisterUnit(Model, "days");
	auto Dimensionless = RegisterUnit(Model);

	auto MyGroup = RegisterParameterGroup(Model, "My␣group");

	auto X = RegisterInput(Model, "X");
	auto A = RegisterParameterDouble(Model, MyGroup, "A", Dimensionless, 1.0);

	auto AX = RegisterEquation(Model, "AX", Dimensionless);

	EQUATION(Model, AX,
		return PARAMETER(A) * INPUT(X);
	)
}
```

Example 7. mytestparameters.dat

```
parameters:
"Timesteps":
10

"Start date":
2019-02-22

"A":
5.0
```

Example 8. mytestinputs.dat

```
timesteps:
10

inputs:
"X":
1 2 3 4 5 6 7 8 9 10
```

Example 5 shows a main function that you can usually keep unchanged throughout model development. The only thing you may want to change in it is the debug printout (`PrintResultSeries`) at the end, to print out different result series. Your main model development will happen inside `mymodel.h` (which you can name anything you like). You will also have to update your parameter and input files as the model requirements change. The parameter file can be autogenerated, something that is explained in the tutorials.

Note that the "Timesteps" and "Start date" parameters are automatically registered with every new model, so you don't have to register them.

4.1.1 A note on visualisation of model results

During model development it can be pretty unsatisfactory to just look at the numbers printed out by `PrintResultSeries` in order to determine if the model performs as expected. It can be a good idea to set up an MobiView (or python wrapper) - compatible model application early. We will not provide an in-depth description of how to do that in this document, but it should not be too difficult to follow the example of existing model applications. See also some notes about this on the front page of the Mobius github repository.

4.2 Entity registration, handles, and equation bodies

The `mobius_model` object contains lists of various model entities (see Table 1 for an explanaition) and information about these. It is the model developer's job to register all the entities and provide the needed information for each one.

Example 9. Example of entity registration.

```
auto X = RegisterInput(Model, "X");
auto A = RegisterParameterDouble(Model, System, "A", Dimensionless, 1.0);
```

The input "X" is tied to the model and is given the name "X". The parameter (of type double) is tied to the Model, the parameter group System, given the name "A", the unit Dimensionless, and the default value 1.0. A full model building API describing all the registration functions is given in Section 6.1.

Each registration function returns a handle. In Example 9, the handles are `X` and `A` (declared with `auto` to signify that we don't care about their exact type). Handles are "tickets" one can use to talk about an entity one has already registered. If you provide the handle `A` in the context of extracting a parameter value, the model will know that you are talking about the parameter "A" that you registered earlier.

Every equation in the model also receives an equation body, and inside the equation body one can extract the values of registered parameters, inputs and equations by referring to them by their handle.

Example 10. The equation "AX" references the value of the parameter "A" and the input "X" by using the handles returned from the registration of those entities.

```
auto AX = RegisterEquation(Model, "AX", Dimensionless);
EQUATION(Model, AX,
       return PARAMETER(A) * INPUT(X);
)
```

Note 3. If you are confused by the syntax of declaring an equation body because it does not look like valid C++, don't worry. `EQUATION` is a macro that expands to declaring a C++11 lambda that is then stored in the `Model` object. The macro is needed to hide some of the underlying functionality, like what object the `PARAMETER` and `INPUT` accessors read their values from, so that the model builder does not have to type too much repetitive code all the time.

4.3 Implementing an example mathematical model in Mobius

The SimplyP model [2] uses the following very simple snow module.

Symbol	Type	Units	Name	Default	Min	Max
$D_{snow,0}$	parameter	mm	Initial snow depth	0.0	0.0	10000.0
f_{DDSM}	parameter	$mm\,d^{-1}\,°C^{-1}$	Degree-day factor for snowmelt	2.74	1.6	6
$Pptn$	input timeseries	$mm\,d^{-1}$	Precipitation			
T_{air}	input timeseries	$°C$	Air temperature			
P_{snow}	state variable	$mm\,d^{-1}$	Precipitation falling as snow			
P_{rain}	state variable	$mm\,d^{-1}$	Precipitation falling as rain			
P_{melt}	state variable	$mm\,d^{-1}$	Snow melt			
D_{snow}	state variable	mm	Snow depth (water equivalents)			
P	state variable	$mm\,d^{-1}$	Hydrological input to soil			

Table 2: SimplyP snow parameters and state variables

The state variables at a given timestep t are given by the following equations

$$P_{snow,t} = \begin{cases} 0 & \text{if } T_{air,t} > 0 \\ Pptn_t & \text{otherwise.} \end{cases}$$

$$P_{rain,t} = \begin{cases} 0 & \text{if } T_{air,t} <= 0 \\ Pptn_t & \text{otherwise.} \end{cases}$$

$$P_{melt,t} = \min(D_{snow,t-1},\, \max(0,\, f_{DDSM} T_{air,t}))$$

$$D_{snow,t} = D_{snow,t-1} + P_{snow,t} - P_{melt,t}$$

$$P_t = P_{rain,t} + P_{melt,t}$$

Say that one also wants a different initial snow depth and degree-day factor for each reach (subcatchment). First one has to register all of the model entities.

Example 11. Registration of model entities for the SimplyP snow module.

```
auto Mm              = RegisterUnit(Model, "mm");
auto MmPerDegreePerDay = RegisterUnit(Model, "mm/°C/day");
auto MmPerDay        = RegisterUnit(Model, "mm/day");

auto Reach = RegisterIndexSetBranched(Model, "Reaches");

auto Snow = RegisterParameterGroup(Model, "Snow", Reach);

auto InitialSnowDepth     = RegisterParameterDouble(Model, Snow, "Initial␣snow␣depth"
    , Mm, 0.0, 0.0, 10000.0);
auto DegreeDayFactorSnowmelt = RegisterParameterDouble(Model, Snow, "Degree-day␣
    factor␣for␣snowmelt", MmPerDegreePerDay, 2.74, 1.6, 6.0);

auto Precipitation = RegisterInput(Model, "Precipitation");
auto AirTemperature = RegisterInput(Model, "Air␣temperature");

auto PrecipitationFallingAsSnow = RegisterEquation(Model, "Precipitation␣falling␣as␣
    snow", MmPerDay);
auto PrecipitationFallingAsRain = RegisterEquation(Model, "Precipitation␣falling␣as␣
    rain", MmPerDay);
auto PotentialDailySnowmelt  = RegisterEquation(Model, "Potential␣daily␣snowmelt",
    MmPerDay);
auto SnowMelt                = RegisterEquation(Model, "Snow␣melt", MmPerDay);
auto SnowDepth               = RegisterEquation(Model, "Snow␣depth", Mm);
auto HydrologicalInputToSoilBox = RegisterEquation(Model, "Hydrological␣input␣to␣soil
    ␣box", MmPerDay);

SetInitialValue(Model, SnowDepth, InitialSnowDepth);
```

Note that we also use the `SetInitialValue` procedure to tell the model that "Snow depth" should have initial value "Initial snow depth". The value of "Initial snow depth" is not determined at this point, but it will be determined before one runs the model (typically by reading a parameter file), and the framework will remember that it should set this value as the initial value of "Snow depth" before the run starts. We have also added the state variable "Potential daily snowmelt" as a partial calculation used in "Snow melt", however that is not strictly necessary. Next one provides the equation bodies.

Example 12. Equation bodies for the SimplyP snow module.

```
EQUATION(Model, PrecipitationFallingAsSnow,
       double precip = INPUT(Precipitation);
       if(INPUT(AirTemperature) <= 0) { return precip; }
       else { return 0.0; }
)

EQUATION(Model, PrecipitationFallingAsRain,
       double precip = INPUT(Precipitation);
       if(INPUT(AirTemperature) > 0) { return precip; }
       else { return 0.0; }
)

EQUATION(Model, PotentialDailySnowmelt,
       return Max(0.0, PARAMETER(DegreeDayFactorSnowmelt) * INPUT(AirTemperature));
)

EQUATION(Model, SnowMelt,
       return Min(LAST_RESULT(SnowDepth), RESULT(PotentialDailySnowmelt));
)

EQUATION(Model, SnowDepth,
       return LAST_RESULT(SnowDepth) + RESULT(PrecipitationFallingAsSnow) - RESULT(
           SnowMelt);
)

EQUATION(Model, HydrologicalInputToSoilBox,
       return RESULT(SnowMelt) + RESULT(PrecipitationFallingAsRain);
)
```

In Example 16 we see that `RESULT` refers to the value of another equation from the current timestep, while `LAST_RESULT` refers to the value from the previous timestep. Looking at the model result structure that is printed using the `PrintResultStructure` procedure, we see the following.

Example 13. Batch structure for the SimplyP snow module.

```
[]
       -----
       Precipitation falling as snow
       Precipitation falling as rain
       -----

[Reaches]
       -----
       Potential daily snowmelt
       Snow melt
       Snow depth
       Hydrological input to soil box
       -----
```

The framework automatically determined that "Potential daily snowmelt" has to index over "Reaches" since it references the parameter "Degree-day factor for snowmelt", which we declared to be part of the "Snow" group, which indexes over "Reaches". Next, "Snow melt" does not reference any parameters, but it does reference "Potential daily snowmelt", and so it too has to

index over "Reaches". "Snow depth" inherits a "Reaches" dependency both from "Snow melt" and from its intial value parameter "Initial snow depth".

Even if we had declared the equations in a different order, the framework would have been able to put them in an order that makes sense. Every equation will be evaluated after any other equation that it accesses the current-timestep result of (i.e. using the `RESULT` accessor). Previous-timestep result accesses (`LAST_RESULT`) do not have any impact on evaluation order. In this case, the equations "Snow depth" and "Hydrological input to soil box" do not depend on (the current-day value of) each other, and so they could have been placed by the framework in any order respective to one another.

In this example we used an input file that does not declare any index set dependencies for "Air temperature" or "Precipitation". If we had instead used an input file where these two timeseries are given per index in "Reaches", we would get the following batch structure.

Example 14. Batch structure for the SimplyP snow module if the user provided precipitation and air temperature timeseries per reach.

```
[Reaches]
       -----
       Precipitation falling as snow
       Precipitation falling as rain
       Potential daily snowmelt
       Snow melt
       Snow depth
       Hydrological input to soil box
       -----
```

Because of the dynamic batch structure, it can change if you start adding other equations later. For instance, the framework may decide to interleave new equations between the existing ones, or move some of the existing ones to a later batch, but correctness is always preserved.

4.3.1 Warning about value accesses inside conditionals

You may have noticed that in the "Precipitation falling as snow" example we extracted the value of precipitation outside the `if` conditional.

Example 15. Correct. Extracting values outside conditionals.

```
EQUATION(Model, PrecipitationFallingAsSnow,
       double precip = INPUT(Precipitation);
       if(INPUT(AirTemperature) <= 0) { return precip; }
       else { return 0.0; }
)
```

This has to do with a limiting factor of the implementation of the framework. It is related to how it registers what equations depend on what other entities. When the model is finalized, the framework evaluates each equation once in a certain state to determine what entities it referenced. Since the equation may go down either branch (but not both) of the conditional `if` during this registration phase, it may miss value accesses that happened in the other branch. You can however access a value inside a conditional if you access it inside both branches.

Example 16. WRONG. DO NOT DO THIS! Extracting values inside conditionals.

```
EQUATION(Model, PrecipitationFallingAsSnow,
       if(INPUT(AirTemperature) <= 0) { return INPUT(Precipitation); }
       else { return 0.0; }
)
```

The general rule is that if you access an entity value at any point in the equation body, you have to make sure that this entity value is accessed regardless of the state of any other values.

Note 4. This means that you also have to make sure that an equation does not (potentially) return before it has accessed all the values it could access!

4.3.2 What to do inside of or outside of equation bodies

Generally you should try to limit what kind of expressions you use inside equation bodies. Things to do inside `EQUATION` bodies:

i Access values using `PARAMETER`, `INPUT`, `RESULT`, `LAST_RESULT`, or any of the more advanced accesses described in Section 4.5.

ii Simple control flow like `for` or `if` (but remember the warning about accesses inside conditionals above!!).

iii Calling functions (that don't have side effects) that do mathematical computations on values, like `exp`, `pow`, `sin` etc.

iv Always returning a value of type double (if you do not do this, you will get a compilation error).

v You can refer to variables that are declared outside the equation body, but you should generally limit this to the entity handles you get from entity registration. Any variable you reference inside the equation body will be copied (by value) using the rules of C++11 lambda captures.

Things to do outside equation bodies, in the registration code.

i Registering entities using the registration procedures.

ii Registering additional relationships between model entities, such as with the `SetInitialValue` procedure, or `SetSolver` (explained later).

Remember that you always have to register an entity before you refer to it inside an equation body.

4.4 Advanced equation types

4.4.1 Cumulation equations

Cumulation equations allow you to sum up the result of another equation over a given index set. Cumulation equations are unique in that you don't provide a body for them. Instead, the framework infers the body of the equation from the data provided in the registration.

Example 17. Declaring a cumulative equation that sum up the value of the `DiffuseFlowOutput` of each landscape unit, determining the total diffuse flow output of the subcatchment.

```
auto TotalDiffuseFlowOutput = RegisterEquationCumulative(Model, "Total␣diffuse␣flow␣
    output", DiffuseFlowOutput, LandscapeUnits);
//NOTE: Don't provide an equation body for TotalDiffuseFlowOutput!
```

A cumulative equation will get the same unit as the equation it cumulates. It will also get the same index set dependencies *except* for the index set it cumulates over. In Example 17, if "Diffuse flow output" indexes over "Reaches" and "Landscape units", then "Total diffuse flow output" indexes over "Reaches" only.

4.4.2 ODE equations and solvers

A solver is a model entity of its own. A solver is registered with an integration method and some configuration values to the integration method. One can then add an equation to the solver batch using `SetSolver` (see example below). All equations with the same solver will be treated as one batch. This means that the solver inherits all the index set dependencies of all the equations in the solver. All the equations will then be evaluated for all of these indexes regardless of whether they had all the index sets as dependencies individually. The solver can (and usually will) evaluate the entire set of equations more than one time per timestep (and index combination). For instance it is not uncommon that a solver evaluates all the equations for 10 or more sub-timesteps per timestep. Moreover, some integration methods (like higher-order Runge-Kutta methods) require multiple evaluations per sub-timestep.

In the next example we have a predator-prey system subject to the Lotka-Volterra equations

$$\dot{x} = \alpha x - \beta xy$$
$$\dot{y} = \delta xy - \gamma y$$

Where x is the prey population, y is the predator population and α, β, γ and δ are parameters. This can be implemented as follows

Example 18. Implementing the Lotka-Volterra system in Mobius. Parameter and unit registration is omitted.

```
auto LVSolver = RegisterSolver(Model, "Lotka-Volterra␣solver", 0.1, IncaDascru); //It
     is not important what you name the solver.
auto Predation = RegisterEquation(Model, "Predation", Individuals);
SetSolver(Model, Predation, LVSolver);
auto PredatorGrowth = RegisterEquation(Model, "Predator␣growth", Individuals);
SetSolver(Model, PredatorGrowth, LVSolver);
auto PreyPopulation = RegisterEquationODE(Model, "Prey␣population", Individuals);
SetSolver(Model, PreyPopulation, LVSolver);
SetInitialValue(Model, PreyPopulation, InitialPreyPopulation); //
    InitialPreyPopulation is a parameter
auto PredatorPopulation = RegisterEquationODE(Model, "Predator␣population",
    Individuals);
SetSolver(Model, PredatorPopulation, LVSolver);
SetInitialValue(Model, PredatorPopulation, InitialPredatorPopulation); //
    InitialPredatorPopulation is a parameter

EQUATION(Model, Predation,
       return PARAMETER(PredationBaseRate) * RESULT(PreyPopulation) * RESULT(
           PredatorPopulation); //PredationBaseRate is beta
)

EQUATION(Model, PredatorGrowth,
       return PARAMETER(PredatorBaseGrowhtRate) * RESULT(PreyPopulation) * RESULT(
           PredatorPopulation); //PredatorBaseGrowthRate is delta
)

EQUATION(Model, PreyPopulation,
       return PARAMETER(PreyBirthRate) * RESULT(PreyPopulation) - RESULT(Predation);
            //PreyBirthRate is alpha
)

EQUATION(Model, PredatorPopulation,
       return -PARAMETER(PredatorDeathRate) * RESULT(PredatorPopulation) + RESULT(
           PredatorGrowth); //PredatorDeathRate is gamma
)
```

There are a couple of things to note from Example 18.

i You can put both regular and ODE equations on a solver. The regular equations will serve as partial computations that can be used by the ODE equations. In the example, it is a little excessive to factor out `Predation` and `PredatorGrowth` as their own equations, but one can easily imagine more complicated examples where a big computation can be reused by many ODE equations.

ii Instead of returning the value of its state variable, an ODE equation has to return the time-derivative of its state variable. The integrator will then integrate the equation system to calculate the actual values of the ODE state variables.

iii The integration of the ODE equations starts every timestep at the end-of-timestep value from the previous timestep, and then integrates the system over one timestep.

For each time the solver evaluates its batch (i.e. typically many times per timestep), the regular equations are evaluated in order before the ODE equations. Within an ODE-system, a reference

to a `RESULT` from the current solver batch will give you the intermediate sub-timestep value that the integrator is currently at. The `RESULT` of an ODE equation will refer to the (integrated) value of the state variable, not the value returned by the ODE equation, which was the time-derivative of the state variable.

If you refer to the `RESULT` value of a solver batch equation from an equation belonging to a different batch (i.e. it was not put on the same solver or any solver at all), you will get the end-of timestep value of that state variable. This is also the value that will be stored in the result timeseries that you can extract after a model run.

Note 5. While for regular equations the framework sorts them so that an equation is evaluated after another equation if it uses the value of the other equation, this is not the case if the other equation is an ODE equation. In that case the only requirement is that the referencing equation is put in the same batch (if it was registered with the same solver) or a batch later than the ODE equation. This is again because the ODE equation does not return a state value, but a derivative. All the state variables of the ODEs within a batch are advanced "simultaneously" during the integration step. See more about this in Section 4.8.

Which integration methods are available is subject to expansion. It is better that you look for them yourself in the source code, such as in `mobius_solver.h` or `boost_solvers.h`. Usually, `IncaDascru` (an adaptive 4th order Runge-Kutta) performs pretty well unless the ODE system is stiff. In case of a stiff system, you could use an implicit solver like `BoostRosenbrock4`, but it is significantly slower and should only be used if absolutely needed.

4.4.3 Initial value equations

Initial value equations are special equations that are not evaluated during the main model run. Instead they are evaluated once (sometimes over multiple indexes) at the start of the model run to determine the initial value of another state variable. This is useful, because sometimes you don't want the initial value of a state variable to be given by a parameter, but by a computation depending on several parameters or initial values of other state variables.

Example 19. An initial value equation referencing parameters.

```
auto SoilVolume = RegisterEquation(Model, "Soil volume", M3);
auto InitialSoilVolume = RegisterEquationInitialValue(Model, "Initial soil volume",
    M3);
SetInitialValue(Model, SoilVolume, InitialSoilVolume);

EQUATION(Model, InitialSoilVolume,
       return PARAMETER(CatchmentArea) * 1e6 * PARAMETER(InitialSoilDepth);
)
//...
```

Initial value equations can also reference the result of other equations. What this means in practice is that they access the initial value of the other equation, whether that is given by a parameter or another initial value equation. Every equation is always given the initial value of 0 unless an initial value is explicitly specified. When you reference the initial value of another equation, you reference the handle of the equation itself, not the handle of the initial value equation (The initial value equation does not have any storage of its own for the value it generates, it just generates a value that is stored for the equation it is computing the initial value for).

Example 20. Another initial value equation that references the initial value computed in Example 19.

```
auto InitialSoilPMass = RegisterEquation(Model, "Initial␣soil␣P␣mass", Kg);

EQUATION(Model, InitialSoilPMass,
       return PARAMETER(InitialSoilPConcentration) * RESULT(SoilVolume); //NOTE: we
           have to reference SoilVolume rather than InitialSoilVolume here.
)
```

There are a couple of things to be aware of when referencing `RESULT`s inside an initial value equation.

i Initial value equations have their own evaluation order *inside the solver batches of the equations they generate initial values for*. What this means is that if you have two initial value equations where their "parent" equations are in the same batch, you can let the initial value equations have a different dependency order than the parent equations.

ii You can not reference the `RESULT` of an equation that is in a later batch. You can also not reference the `RESULT` of an equation that comes in a previous batch *unless* you use explicit indexing. This is an implementation limitation. If this is something that causes big problems for users, this is something we could try to improve. Note that this is again something you just have to be aware of while building the model, since the batch structure may not be apparent until a large part of the model has come together. You will usually not get error messages if you reference results that are not "available", but you may get a 0 or a result that is misindexed.

4.4.4 Computed parameters (experimental feature)

If you have a value in the model that you know can be computed based on other parameters and will not change during model run, you can have this value as a computed parameter. Register it as a parameter in the usual way, then call `ParameterIsComputedBy`.

Example 21.

```
auto SoilMass = RegisterParameterDouble(Model, Land, "Soil␣mass", KgPerKm2, 1e6);
ParameterIsComputedBy(Model, SoilMass, SoilMassComputation, true);
```

In the above example `SoilMassComputation` is an initial value equation. This initial value equation can only refer to the value of other parameters in its equation body, never the value of other equations or inputs. The last boolean argument to the `ParameterIsComputedBy` call, if `true`, says that this parameter should never be exposed in parameter files (or be editable in GUIs).

The step to compute parameters is executed right before the step that initializes the initial values of all equations. The initial value equation for the parameter is evaluated once for each instance of the parameter regardless of what index set dependencies the equation itself has. Right now it is not safe for this equation to depend on the value of other computed parameters since we don't sort the computations in order of dependencies for these.

4.5 Advanced value accesses

4.5.1 Special state values

The framework lets you access a few additional variables that tell you something about the state the equation is called under. These are.

i `CURRENT_INDEX(IndexSetHandle)` This will return an `index_t` which tells you the current index of the index set corresponding to the provided `IndexSetHandle`. An `index_t` is a small struct that contains information about what index set and what index it belongs to. In addition to providing the value of the current index, `CURRENT_INDEX` forces the equation to depend on this index set, i.e. it will be evaluated for each index in the index set (otherwise, if it was not evaluated for each index in the index set, it would not make sense to ask what was the current index of that index set during evaluation).

ii `INDEX_COUNT(IndexSetHandle)` This will return an `index_` that is always guaranteed to be one larger than the largest index in the index set corresponding to the provided `IndexSetHandle`. Registers no dependencies.

iii `FIRST_INDEX(IndexSetHandle)` Gives you an `index_t` corresponding to the first index of that index set. Registers no dependencies.

iv `INDEX_NUMBER(IndexSetHandle, Number)` Gives you an `index_t` corresponding to the index of that number. However, be careful when using this that the index is smaller than `INDEX_COUNT(IndexSetHandle)`. Also, it is probably better to use a predeclared index, which guarantees you that what you are referring to has the right index name (see Section 4.5.3 below). Registers no dependencies.

v `CURRENT_DAY_OF_YEAR()` Returns an unsigned integer telling you which day of year it is (in the current model state). January 1st is always day 1. For instance, if the model "Start date" is 1999-1-15, then during the first timestep, the current day of year will be 15.

vi `DAYS_THIS_YEAR()` Returns an unsigned integer that is either 365 (in a normal year) or 366 (in a leap year).

Note 6. The `index_t` structure only contains numeric info about what index set handle it belongs to and the number of the index. However, instead of specifying these directly, it is better to use one of the wrapper macros that allow you to get an index of a specific index set. There are also a couple of overloaded operators that allow you to do size comparisons or increment `index_t`'s.

4.5.2 Explicit indexing and explicit iteration

The most common case is that you don't have to use any explicit `for` loops. Instead the batch structure takes care of the `for` loops for you, evaluating every equation for all combination of indexes it depends on. Moreover, if you want to sum up the result of an equation, you can always use a cumulation equation. There may however be situations where you want to do something more complicated.

Example 22. A simplified version of the percolation output function in PERSiST.

```
EQUATION(Model, PercolationOut,
        double percolationOut = 0.0;

        for(index_t OtherSoil = CURRENT_INDEX(SoilBoxes) + 1; OtherSoil < INDEX_COUNT
            (SoilBoxes); ++OtherSoil)
        {
                double mpi = PARAMETER(Infiltration, OtherSoil);
                double perc = Min(mpi, PARAMETER(PercolationMatrix, OtherSoil) *
                    RESULT(TotalRunoff));

                percolationOut = Min(perc + percolationOut, RESULT(WaterDepth3));
        }
        return percolationOut;
)
```

In Example 22 we are in one of PERSiST's soil layers, and we want to iterate over every soil layer below this one (in this case, the indexes have been declared in order from top to bottom, so "below" is the same as after in the list of indexes), to see how much of the runoff from this soil layer is distributed to the other soil layers. To do this we iterate from `CURRENT_INDEX(SoilBoxes)+1` to `INDEX_COUNT(SoilBoxes)`. We then look up the maximum infiltration to the other soil box by calling `PARAMETER(Infiltration, OtherSoil)`. We are looking up the value of the parameter `Infiltration`, but instead of looking up the value for the current index of `SoilBoxes` using `PARAMETER(Infiltration)`, we provide an explicit index for the other soil box.

There are a few prerequisites for allowing you to do this

i The parameter `Infiltration` has to depend on the index set `SoilBoxes`.

ii Moreover, `SoilBoxes` has to be the immediate index set dependency for `Infiltration`. This means that `Infiltration` is in a parameter group that indexes over `SoilBoxes` directly (as opposed to only having a parent group that indexes over `SoilBoxes`).

Even if you explicitly index only one index set, the parameter can have more than one index set dependency. I.e. its parameter group can be a member of another parameter group which itself has an index set dependency. When you explicitly index only the immediate index set dependency, the index of the higher level index sets is as usual inferred to be the current index of these index sets in the execution state. Moreover, the call will register a dependency for `PercolationOut` on the higher level index set(s) that were automatically inferred, but not on `SoilBoxes`. (Though, `PercolationOut` does get a dependency on `SoilBoxes` any way from the call to `CURRENT_INDEX(SoilBoxes)`).

Example 23. Equation `A` refers to `PARAMETER(B, IX)` in its equation body. Parameter `B` has the dependencies {"Reaches", "Landscape units"}, where "Landscape units" is the immediate dependency, and "Reaches" is a dependency of the parent group of the immediate group. The index `IX` will refer to an index of "Landscape units". The framework determines that `A` has to index over "Reaches".

You can also provide more than one explicit index.

Example 24. Equation `A` refers to `PARAMETER(B, IX, IY)` in its equation body. Parameter `B` has the dependencies {"Reaches", "Landscape units", "Soils"}, where "Soils" is the immediate dependency, "Landscape units" is a dependency of the parent group of the immediate group, and "Reaches" is again a dependency of a parent of that group. The index `IX` will refer to an index of "Landscape units", `IY` of "Soils". The framework determines that `A` has to index over "Reaches".

Example 24 also generalizes to higher numbers of explicit indexes. If you only want to explicitly index one of the higher index sets and let the framework infer the index of the immediate one, you can do the following.

Example 25. Explicitly indexing a higher index only.

```
PARAMETER(PercolationMatrix, OtherSoil, CURRENT_INDEX(SoilBoxes))
```

Explicit indexing of `INPUT`s is currently not available.

Explicit indexing of `RESULT` and `LAST_RESULT` follows some of the same rules as explicit indexing of parameters, however here you have to be much more careful. For one thing, during model development, you can not be sure in which order the index set dependencies of an equation will be in, or even which index set dependencies it will have. This is after all determined by the framework at runtime. You therefore have to check that you guessed the correct order of the dependencies by looking at the equation batch structure (which could change if you for instance change the dependencies of some parameters).

Note 7. You should generally be careful with explicit indexing of `RESULT`s inside equations that are in a solver batch. If you explicitly index a result from the same batch you will either get just the end-of-timestep value (if you indexed an already evaluated index) or a 0 (if you for some reason explicitly indexed the currently evaluated index). This is because the sub-timestep values of the ODE system are not stored out to the main result storage and are discarded after (or during) integration. To give an example, while an expression like `RESULT(WaterDepth, CURRENT_INDEX(Soils))` would give the same value as `RESULT(WaterDepth)` outside a solver (albeit slower), it would give a 0 inside a solver (if `WaterDepth` is also evaluated in the solver). Of course, there should not be any reason to use such an expression in the first place. Explicit indexing of parameters is ok, but see the notes on performance under Section 4.7.

4.5.3 Predeclared indexes

In some cases you may not want the indexes of an index set to be user defined, instead you want the model to determine the indexes. In that case you can call `RequireIndex` one or more times to tell the model what indexes you want in the index set. This call will also return to you an `index_t` value that you can use to explicitly refer to this index inside `EQUATION`s.

Example 26. Explicit indexing using predeclared indexes for the index set with handle `Soils`.

```
index_t DirectRunoff = RequireIndex(Model, Soils, "Direct runoff");
index_t Soilwater = RequireIndex(Model, Soils, "Soil water");
index_t Groundwater = RequireIndex(Model, Soils, "Groundwater");

EQUATION(Model, DirectRunoffVolume,
       return RESULT(WaterDepth, DirectRunoff) * 1000.0;
)
```

The order of these indexes in the index set will be the order they were declared in using `RequireIndex`.

4.5.4 Branch iteration

If you have a branched index set, you may want to use the special connectivity information that is carried by its indexes. You can do this using branch iteration, which iterates over all the branch inputs to the current index of this index set.

Example 27. Iterating over all the branch inputs to the current index of the branched index set with handle `Reach`. Here we sum up the result of `ReachFlow` coming in from each branch.

```
EQUATION(Model, ReachUpstreamFlow,
        double upstreamflow = 0.0;
        FOREACH_INPUT(Reach,
                upstreamflow += RESULT(ReachFlow, *Input);
        )
        return upstreamflow;
)
```

The `FOREACH_INPUT` expression is a macro that expands to an iteration over the branches. The `index_t` of each branch that is iterated over is given by `*Input`.

4.6 Debugging features

Apart from using full C++ debugging techniques, there are a few framework features that may help you to better debug unexpected model behaviour. We have already covered using `PrintResultStructure(Model)` to show you the equation batch structure, which is something you should have turned on at all times during model development. You can also use `PrintParameterStorageStructure(DataSet)` and `PrintInputStorageStructure(DataSet)` to show you the index set dependencies of each parameter and input timeseries.

There are also five flags that you can turn on that will make the model store and print out additional debug information during the model run. These have to be declared above your `#include "mobius.h"`

Example 28. Declaring the debug flags.

```
#define MOBIUS_TIMESTEP_VERBOSITY 1
#define MOBIUS_TEST_FOR_NAN 1
#define MOBIUS_PRINT_TIMING_INFO 1
#define MOBIUS_EQUATION_PROFILING 1
#define MOBIUS_INDEX_BOUNDS_TESTS 1

#include "../../mobius.h"
```

You can turn the debug flags either on or off by setting them to 1 or 0 respectively (or turn them off by removing them entirely).

i `MOBIUS_TIMESTEP_VERBOSITY`. This has 3 levels. If it is set to 1 it will print out the number of the timestep for every timestep. If it is set to 2, it will also print out every time it changes the state of an index set during the iteration. If it is set to 3 it will also print out every time

it finishes the evaluation of an equation (or an integration in the case of a solver), along with the value. This can give you a full sense of the model evaluation like it is described in Example 4 (You should probably turn down the number of timesteps if you want to test the model in this mode).

ii `MOBIUS_TEST_FOR_NAN`. If turned on, this will test every result value to see if it was an inf or nan. In the case of such a value, it will halt the model execution and print out the current timestep, the state of the index sets, and the state of all the parameter, input and result dependencies of the equation that produced the illegal value. This will make the model run a little slower even if it does not encounter an illegal value. The nan test will not work correctly if you compile with the -ffast-math flag. The -ffast-math flag may make the model run a little faster, but the underlying floating point architecture will no longer make nans detectable, which can interfer both with this safeguard and other safeguards you may put in.

iii `MOBIUS_PRINT_TIMING_INFO`. If turned on, this will time and print out how long it took to run the model (both in milliseconds and processor instructions). This only gives very coarse information, but can be useful to detect if something weird happened (does the model suddenly run much slower after you added a new equation?).

iv `MOBIUS_EQUATION_PROFILING`. This will give you very fine-grained information about how many processor ticks every equation evaluation took on average, and how many times each equation was evaluated. Note that the time to evaluate an equation is not always the same as the time to produce a result value since a solver may evaluate an equation many times to produce a result value for it. See Section 4.7 about tips on how to improve the speed of the model. Having this flag turned on will make the framework itself a little slower.

v `MOBIUS_INDEX_BOUNDS_TESTS`. This should probably be turned on as often as possible during model development. It will catch if during explicit indexing you provided an `index_t` that was out of bounds for this index set. It will also give you an error if you use it to index a different index set than it was created from. Having this flag turned on will make the model run a little slower.

4.7 Tips on model performance (speed)

This section will contain a few general tips on how to make the model run as fast as possible.

One important thing you can do is to turn on the `MOBIUS_EQUATION_PROFILING` flag (see the previous section) to see which individual equations are slow. You should then weigh the importance of improving the speed of an equation by the number of times it is evaluated. Equations with many index set dependencies are evaluated more often than equations with few index set dependencies, and equations in solvers are evaluated *many* more times than equations that are not on solvers.

The following things are common causes of slow equations:

i Calling transcendental functions from the C++ (C) standard math library, like `exp`, `pow` or `sin`. The standard implementation of these is often slow because they have to satisfy a large range of criteria. If you know that you only need implementations that are correct in a short value range (i.e. $[0, 2\pi)$) or need a smaller precisional correctness (say only 4 digits), then you may be able to find implementations that are much faster. You could also look into implementing that yourself, but this is a big topic of its own, and will not be covered here. Of course, it may often not be worth it going into the trouble of replacing the implementation of these.

ii Explicit indexing (e.g. calling `RESULT(Handle, Index)`) is always slower than implicit indexing (e.g. `RESULT(Handle)`). This is because the model keeps all the values you need for implicitly indexed values in a quick buffer that it updates for you using a heavily optimized system. On the other hand, if you explicitly index a value, the model has to first determine where that value is stored using an index calculation, and then go back to main storage to retrieve it (risking cache misses).

Since equations on solvers are much more expensive than equations outside solvers, you may sometimes want to see if you can factor a calculation made in a solver equation out to a separate equation that can be evaluated before the solver is run.

Example 29. Instead of doing the reach flow in one calculation,

```
EQUATION(Model, ReachFlow,      //ReachFlow is an ODE equation
        double reachflowinput = 0.0;
        FOREACH_INPUT(Reach,
                reachflowinput += RESULT(ReachFlow, *Input);
        )
        reachflowinput += RESULT(TotalDiffuseFlowOutput);

        return ( reachflowinput - RESULT(ReachFlow) ) / ( RESULT(ReachTimeConstant) *
             (1.0 - PARAMETER(B)));
)
```

separate out the expensive bit that uses explicit indexing (and don't run the expensive bit on the solver).

```
EQUATION(Model, ReachFlowInput,        //ReachFlowInput should not run on the solver
        double upstreamflow = 0.0;
        FOREACH_INPUT(Reach,
                upstreamflow += RESULT(ReachFlow, *Input);
        )
        return upstreamflow + RESULT(TotalDiffuseFlowOutput);
)

EQUATION(Model, ReachFlow,           //ReachFlow should still run on the solver
        return (RESULT(ReachFlowInput) - RESULT(ReachFlow) ) / ( RESULT(
            ReachTimeConstant) * (1.0 - PARAMETER(B)));
)
```

Cumulation equations are also always faster than explicit for loops, because the framework can use some tricks to determine how to look up the indexes of values accessed by a cumulation equation. Only use explicit for loops if you have to.

4.8 Determining the batch structure of advanced models

Note 8. This section is very technical, and so you could probably skip it until your model does something completely unexpected and you have to figure out why.

Every equation has three types of dependencies that are important to model structure.

i Index set dependencies are the index sets that the equation should index over (i.e. be evaluated for every combination of indexes of). Again, remember that unlike for parameters, index sets can only appear once in the dependency list of an equation.

ii Direct result dependencies are other equations of which this equation references the `RESULT` of (using implicit indexing: `RESULT(Handle)`).

iii Cross index result dependencies are other equations of which this equation references the `RESULT` of using explicit indexing (e.g. `RESULT(Handle, Index)`).

When processing information about equation dependencies during model finalization (in the `EndModelDefinition` procedure), every equation gets its index set dependencies and result dependencies using the following rules:

i A cumulation equation has the equation it cumulates as a direct result dependency (this is an exception to the description of direct result dependencies above). It also gets all the index set dependencies of the equation it cumulates *minus* the index set it cumulates over.

ii An equation gets all the index set dependencies of all the parameters and inputs it references. If the reference uses explicit indexing, it will not get a dependency for the index sets that are explicitly indexed. It will also get a dependency on index sets it calls `CURRENT_INDEX` or `FOREACH_INDEX` on.

iii An equation gets all the index set dependencies of its initial value parameter or initial value equation if it has one.

iv An equation gets all the index set dependencies of all the other equations it references the `RESULT` or `LAST_RESULT` of. The same rules for explicit indexing hold as for parameters. Note that this creates a large directed dependency graph between equations, where index set dependencies propagate down all the paths of the graph. Thus an equation can inherit an index set dependency not only from an equation it directly accesses the result of, but also one that was accessed by that equation and so on.

The list of equations (excluding initial value equations) are processed into a batch structure obeying the following rules:

i All equations that were put on a solver have to belong to the same solver batch, and no other equations can belong to this batch. The index set dependencies of the batch is the union of the index set dependencies of all its equations. Similarly, the direct result (or cross-index) dependencies of a solver are the union of all the direct (or cross-index) result dependencies of its equations.

ii Non-ode equations in a solver are sorted in order so that a given non-ode-equation comes after any other non-ode equations that it has a direct result dependency on. ODE equations are placed after the non-ode equations in any order (in practice the order they were declared in).

iii Solver batches have to be placed after any equations (not belonging to itself) that it has a direct result dependency on, and in the same batch group as or after any it has a cross-index result dependency on.

iv If two non-solver equations that have the same index set dependencies are placed next to each other, they are merged into a batch.

v A non-solver equation has to be placed after any other equation it has a direct result dependency on. It has to be placed in the same batch group as or after any equation it has a cross-index result dependency on.

vi If a solver batch is next to one or more non-solver batches (or different solver batches) with the same index set dependencies, they form a batch group.

The rules above do not determine a unique structure. To improve the run speed of the model, the framework will also try to move batches and equations around (without breaking the rules) so that there are as few batches and batch groups as possible in the structure. I.e. it will prefer to place equations with the same index set dependencies next to each other. The algorithm we use is not guaranteed to find the most optimal structure (indeed, we are not even sure if such an algorithm would be polynomial or not), but it does produce pretty good results for the particular models we have built so far. The reason that such an minimalization of the amount of batch groups is a good idea is that there is some overhead associated with every time the model running code starts evaluating a new batch group.

During model creation you may accidentally specify equations that have direct result dependencies on each other in a circular way. This means that there is no possible way for the framework to place them in an order where each equation has access to all the result values it needs. The framework will detect this and print an error. Here ODE equations are exceptional in that any equation in the same solver can have a direct dependency on the result of an ODE equation regardless of circular dependencies.

This can sometimes be tricky if a solver is involved. Equation A can not depend on any equation in solver batch B if any equation at all in B (even those that A does not depend on) has a dependency on A. Just remember that the framework can never sort A into the middle of B as it has to evaluate all of B in one go.

For each batch, the order of evaluating the initial value of each equation (either by reading it from a parameter, setting it to a constant, or evaluating an initial value equation) is given its own evaluation order based on the dependencies of the initial value equations.

Note 9. The description given here does not give all the implementation details. To get a complete understanding you probably have to look up the implementation of the `EndModelDefinition` function in `mobius_model.cpp`.

5 Application building API

We logically separate application building from model building. By application building we mean the process of building a program that can allocate a new model, allocate a dataset for it, put values in the dataset, choose how and when to run the model using the given parameters and inputs, extract result values and process them, and so on. By model building we mean the process of filling a given model object with information about the model structure, such as what parameters and equations it has, and what the equation bodies are. It is recommended to separate out the model building to its own procedure, because then the model can be reused by different applications.

5.1 Data types

5.2 Procedures

BeginModelDefinition	Application building procedure.
Arguments:	
`const char *Name`	(optional) The name of the model.
`const char *Version`	(optional) The version of the model.
Returns:	
A `mobius_model*` pointer to a newly created model object.	
Description:	
Heap-allocates a new model object. The model object can be freed using `delete`.	

ReadInputDependenciesFromFile	Application building procedure
Arguments:	
`mobius_model *Model`	Pointer to the model object.
`const char *Filename`	Name of an input file (of the .dat format).
Description:	
If you wish to use an input file you have to let the model read the input dependencies before calling `EndModelDefinition` so that it can properly determine the model structure. This has to happen after the inputs are registered. You then have to read the actual input values into a dataset using `ReadInputsFromFile` after finalizing the model. The current procedure also registers any additional timeseries that are present in the input file.	

EndModelDefinition	Application building procedure.
Arguments:	
`mobius_model *Model`	Pointer to the model object.
Description:	
Finalizes the model object. You can no longer call model building procedures on it, and you should not try to make any further changes to it in general. This procedure generates the final equation batch structure of the model according to the rules given in Section 4.8.	

GenerateDataSet	Application building procedure
Arguments:	
`mobius_model *Model`	Pointer to the model object.
Returns:	
A `mobius_data_set*` pointer to a newly created dataset object.	
Description:	
Heap-allocates a new dataset. The dataset can be freed using `delete`. You can have multiple datasets per model. You have to keep the model object alive as long as you want to use the dataset (i.e. don't `delete` the model before you are finished with the dataset). You can only generate a dataset from a model that has been finalized using `EndModelDefinition`.	

ReadParametersFromFile	Application building procedure
Arguments:	
`mobius_data_set *DataSet`	Pointer to the dataset object.
`const char *Filename`	The name of the parameter file (of the .dat format)
Description:	
Reads the indexes of each index set and the values of all the parameters from a parameter file into the dataset.	

ReadInputsFromFile	Application building procedure
Arguments:	
`mobius_data_set *DataSet`	Pointer to the dataset object.
`const char *Filename`	The name of the input file (of the .dat format)
Description:	
Reads the input timeseries from the file into the dataset. This can only be called if all the index sets registered in the model has been given indexes in the dataset (either by calling `ReadParametersFromFile` or by calling `SetIndexes`).	

RunModel	Appication building procedure
Arguments:	
`mobius_data_set *DataSet`	Pointer to the dataset object.
Description:	
Attempts to run the model (that the dataset was generated from) with the parameter and input values given in this dataset. The start date is read from the "Start date" parameter, and the number of timesteps is read from the "Timesteps" parameter. All result timeseries are stored in the dataset.	

There are a few more procedures that we will try to document soon.

6 Model building API

6.1 Registration procedures

The registration procedures can only be called on a `mobius_model` object that has not been finalized using `EndModelDefinition`.

RegisterUnit	Registration procedure
Arguments:	
`mobius_model *Model`	Pointer to the model object.
`const char *Name`	(optional) The name of the unit.
Returns:	
A `unit_h` handle to the newly registered unit.	
Description:	
Register a new unit with the model. If the name is omitted, it is assumed to be "dimensionless".	

RegisterIndexSet — Registration procedure

Arguments:

`mobius_model *Model`	Pointer to the model object.
`const char *Name`	The full name of the index set.

Returns:
An `index_set_h` handle to the newly registered index set.

Description:
Register a new (basic) index set with the model.

RegisterIndexSetBranched — Registration procedure

Arguments:

`mobius_model *Model`	Pointer to the model object.
`const char *Name`	The full name of the index set.

Returns:
An `index_set_h` handle to the newly registered index set.

Description:
Register a new branched index set with the model.

RequireIndex — Registration procedure

Arguments:

`mobius_model *Model`	Pointer to the model object.
`index_set_h IndexSet`	Handle to an index set.
`const char *Name`	The name of the index.

Returns:
An `index_t` giving the number of the new index in the index set.

Description:
Require the index set to have this index. If an index set has one or more required indexes, it can only have those indexes and not user-defined ones. Can only be called on basic index sets, not branched index sets.

RegisterParameterGroup — Registration procedure

Arguments:

`mobius_model *Model`	Pointer to the model object.
`const char *Name`	The full name of the parameter group.
`index_set_h IndexSet`	(optional) Handle to an index set that the group should index over.

Returns:
An `parameter_group_h` handle to the newly registered parameter group.

Description:
Register a new parameter group with the model.

SetParentGroup — Registration procedure

Arguments:

`mobius_model *Model`	Pointer to the model object.
`parameter_group_h Child`	A handle to the child group.
`parameter_group_h Parent`	A handle to the parent group.

Returns:

`void`

Description:

Set `Parent` to be the parent group of `Child`. Every parameter in `Child` will index both over the index set of `Child` and the index set of `Parent`.

RegisterInput — Registration procedure

Arguments:

`mobius_model *Model`	Pointer to the model object.
`const char *Name`	The full name of the input.
`unit_h Unit`	(optional) A unit for the input.

Returns:

An `input_h` handle to the newly registered input timeseries.

Description:

Register a new input timeseries with the model.

RegisterParameterDouble — Registration procedure

Arguments:

`mobius_model *Model`	Pointer to the model object.
`parameter_group_h ParameterGroup`	Handle to the parameter group that the parameter should be put in.
`const char *Name`	The full name of the parameter.
`unit_h Unit`	Handle to the unit of the parameter.
`double Default`	The default value of the parameter.
`double Min`	(optional) The minimum value of the parameter.
`double Max`	(optional) The maximum value of the parameter.
`const char *Description`	(optional) A long-form description of the parameter.

Returns:

A `parameter_double_h` handle to the newly registered parameter.

Description:

Register a new parameter of type double with the model.

RegisterParameterUInt

Registration procedure

Arguments:

`mobius_model *Model`	Pointer to the model object.
`parameter_group_h ParameterGroup`	Handle to the parameter group that the parameter should be put in.
`const char *Name`	The full name of the parameter.
`unit_h Unit`	Handle to the unit of the parameter.
`u64 Default`	The default value of the parameter.
`u64 Min`	(optional) The minimum value of the parameter.
`u64 Max`	(optional) The maximum value of the parameter.
`const char *Description`	(optional) A long-form description of the parameter.

Returns:

A `parameter_uint_h` handle to the newly registered parameter.

Description:

Register a new parameter of type uint with the model.

RegisterParameterBool

Registration procedure

Arguments:

`mobius_model *Model`	Pointer to the model object
`parameter_group_h ParameterGroup`	Handle to the parameter group that the parameter should be put in
`const char *Name`	The full name of the parameter
`bool Default`	The default value of the parameter
`const char *Description`	(optional) A long-form description of the parameter

Returns:

A `parameter_bool_h` handle to the newly registered parameter.

Description:

Register a new parameter of type bool with the model

RegisterParameterDate

Registration procedure

Arguments:

`mobius_model *Model`	Pointer to the model object.
`parameter_group_h ParameterGroup`	Handle to the parameter group that the parameter should be put in.
`const char *Name`	The full name of the parameter.
`const char *Default`	The default value of the parameter. Must be on the format "y-m-d".
`const char *Min`	(optional) The minimum value of the parameter. Must be on the form "y-m-d".
`const char *Max`	(optional) The maximum value of the parameter. Must be on the form "y-m-d".
`const char *Description`	(optional) A long-form description of the parameter.

Returns:

A `parameter_time_h` handle to the newly registered parameter.

Description:

Register a new parameter of type date(time) with the model.

RegisterEquation — Registration procedure

Arguments:

`mobius_model *Model`	Pointer to the model object.
`const char *Name`	The full name of the equation.
`unit_h Unit`	Handle to the unit of the equation.

Returns:

A `equation_h` handle to the newly registered equation.

Description:

Register a new equation with the model. The is is a basic equation. The equation body has to be provided separately using the `EQUATION` macro.

RegisterEquationODE — Registration procedure

Arguments:

`mobius_model *Model`	Pointer to the model object.
`const char *Name`	The full name of the equation.
`unit_h Unit`	Handle to the unit of the equation.

Returns:

A `equation_h` handle to the newly registered equation.

Description:

Register a new equation of type ODE with the model. The equation body has to be provided separately using the `EQUATION` macro.

RegisterEquationInitialValue — Registration procedure

Arguments:

`mobius_model *Model`	Pointer to the model object.
`const char *Name`	The full name of the equation.
`unit_h Unit`	Handle to the unit of the equation.

Returns:

A `equation_h` handle to the newly registered equation.

Description:

Register a new initial value equation with the model. The equation body has to be provided separately using the `EQUATION` macro.

RegisterEquationCumulative — Registration procedure

Arguments:

`mobius_model *Model`	Pointer to the model object.
`const char *Name`	The full name of the equation.
`equation_h Cumulates`	Handle to the equation that should be cumulated (can not be an initial value equation).
`index_set_h CumulatesOver`	Handle to the index set that should be cumulated over.
`parameter_double_h Weight`	(optional) A weight parameter for the cumulation.

Returns:

A `equation_h` handle to the newly registered equation.

Description:

Register a new initial value equation with the model. No equation body should be provided. If a weight is provided it has to index over the `CumulatesOver` index set. For the purpose of the cumulation, the weight is scaled so that it sums to 1.

SetInitialValue — Registration procedure

Arguments:

`mobius_model *Model`	Pointer to the model object.
`equation_h Equation`	The equation to set the initial value for.
`double Value`	A constant value to set as the initial value.

Returns:

`void`.

Description:

Set the initial value of an equation to be a constant.

SetInitialValue — Registration procedure

Arguments:

`mobius_model *Model`	Pointer to the model object.
`equation_h Equation`	The equation to set the initial value for.
`parameter_double_h InitialValue`	A handle to a parameter of type double.

Returns:

`void`.

Description:

Set the initial value of an equation to be determined by a parameter.

SetInitialValue — Registration procedure

Arguments:

`mobius_model *Model`	Pointer to the model object.
`equation_h Equation`	The equation to set the initial value for.
`equation_h InitialValueEq`	A handle to an initial value equation.

Returns:

`void`.

Description:

Set the initial value of an equation to be computed by an initial value equation.

ResetEveryTimestep — Registration procedure

Arguments:

`mobius_model *Model`	Pointer to the model object.
`equation_h Equation`	Handle to an ODE equation.

Returns:

`void`.

Description:

Tell the model that instead of starting the ODE state variable at the end value from the previous timestep, it should start at 0 each timestep. This is primarily used if you want to integrate some variable over each timestep, such as when you compute mean values.

RegisterSolver — Registration procedure

Arguments:

`mobius_model *Model`	Pointer to the model object.
`const char *Name`	Name of the solver batch (for internal reference).
`double h`	The suggested sub-timestep length of the solver (in the range 0 to 1).
`mobius_solver_setup_function *SetupFunction`	One of the integrator setup functions provided by mobius_solvers.h or boost_solvers.h, or others.
`double RelErr`	(only available for some choices of SetupFunction) The relative error tolerance of the solver.
`double AbsErr`	(only available for some choices of SetupFunction) The absolute error tolerance of the solver.

Returns:

A `solver_h` handle to the newly registered solver.

Description:

Register a new solver batch with the model. You may have to look at the source code to determine which solvers accept error tolerances.

SetSolver — Registration procedure

Arguments:

`mobius_model *Model`	Pointer to the model object.
`equation_h Equation`	Handle to the equation to put in the solver batch.
`solver_h`	The solver batch to put the equation in.

Returns:

`void`.

Description:

Put an equation in a solver batch. Can only be done with basic or ODE equations.

GetXHandle	Registration procedure
Arguments:	
`mobius_model *Model`	Pointer to the model object.
`const char *Name`	The name of an already registered X.
Returns:	
A *x*`_h` handle.	
Description:	
X is either Equation, Input, ParameterGroup, ParameterDouble, ParameterUInt, ParameterBool, ParameterTime, IndexSet, or Solver. Is useful if you want to modularize different parts of your model into several different registration procedures, but need one module to access some of the handles of another one.	

6.2 Equation body declaration

There are a lot of advanced rules governing the use cases of the macros in this section, and so we recommend that you read the documentation above (that means at least all of Section 4, which describes the details. All the equation bodies of the registered equations have to be declared before calling `EndModelDefinition` on the model object.

EQUATION	Equation macro
Arguments:	
`mobius_model *Model`	Pointer to the model object.
`equation_h Equation`	A handle to the equation to declare the body for.
`Decl`	The code for the equation body.

Description:

`EQUATION(Model, Handle, Decl)` expands to

```
SetEquation(Model, Handle,
        [=](value_set_accessor *ValueSet__) -> double {
                Decl
        }
);
```

where `Decl` stands for possibly multiple lines of code of an equation body. These lines of code always have to return a `double`. The `value_set_accessor` is an object that can be used by other macros (such as `PARAMETER` or `RESULT`) inside the `Decl` to extract values.

The following macros are only available inside the `Decl` of an `EQUATION`. This is because they expand to something that refer to the `value_set_accessor` that is given as an argument in the lambda inside the expansion of the `EQUATION` macro.

PARAMETER — Equation macro

Arguments:

`parameter_x_h ParameterHandle`	Handle to a parameter that you wish to extract the value of, where `x` is either `double`, `uint` or `bool`.
`index_t Indexes...`	(optional, varargs) One or more indexes to explicitly index.

Returns:

Either a `double`, a `u64` or a `bool` depending on what type of handle was passed in.

Description:

Returns the value of the parameter, depending on the current state of the index sets. This is a macro that expands to an expression which either extracts a value from the `value_set_accessor` or registers a dependency of the surrounding `Equation` on this parameter, depending on the context of the evaluation.

INPUT — Equation macro

Arguments:

`input_h InputHandle`	Handle to an input that you wish to extract the value of.

Returns:

A `double`.

Description:

Returns the current timestep value of the input timeseries, depending on the current state of the index sets. This is a macro that expands to an expression which either extracts a value from the `value_set_accessor` or registers a dependency of the surrounding `Equation` on this input, depending on the context of the evaluation.

RESULT — Equation macro

Arguments:

`equation_h EquationHandle`	Handle to an equation that you wish to extract the value of.
`index_t Indexes...`	(optional, varargs) One or more indexes to explicitly index.

Returns:

A `double`.

Description:

Returns the value of the current timestep result of the equation, depending on the current state of the index sets. This is a macro that expands to an expression which either extracts a value from the `value_set_accessor` or registers a dependency of the surrounding `Equation` on this equation, depending on the context of the evaluation.

LAST_RESULT — Equation macro

Arguments:

`equation_h EquationHandle`	Handle to an equation that you wish to extract the previous-timestep value of.
`index_t Indexes...`	(optional, varargs) One or more indexes to explicitly index.

Returns:
A `double`.

Description:
Returns the value of the previous timestep result of the equation, depending on the current state of the index sets. This is a macro that expands to an expression which either extracts a value from the `value_set_accessor` or registers a (last-result) dependency of the surrounding `Equation` on this equation, depending on the context of the evaluation.

EARLIER_RESULT — Equation macro

Arguments:

`equation_h EquationHandle`	Handle to an equation that you wish to extract an earlier-timestep value of.
`u64 StepBack`	How many timesteps back you want to go.
`index_t Indexes...`	(optional, varargs) One or more indexes to explicitly index.

Returns:
A `double`.

Description:
Looks up the result of the equation value for the current timestep minus `StepBack` number of timesteps. So if `StepBack=0` you get the current result, if `StepBack=1` you get the last result and so on. This registers the same dependencies as a `LAST_RESULT` lookup.

CURRENT_DAY_OF_YEAR — Equation macro

Arguments:

None.

Returns:
A `u64`.

Description:
Returns the current day of the year in the context of the simulation. January 1st corresponds to a current day of year of 1. This is a macro that expands to an expression that extracts the value from the `value_set_accessor`.

DAYS_THIS_YEAR — Equation macro

Arguments:

None.

Returns:

A `u64`.

Description:

Returns the amount of days this year in the context of the simulation. Is either 365 (in a common year) or 366 (in a leap year). This is a macro that expands to an expression that extracts the value from the `value_set_accessor`.

INDEX_COUNT — Equation macro

Arguments:

`index_set_h IndexSet`	Handle to an index set.

Returns:

A `size_t`.

Description:

Returns the amount of indexes in this index set. This is a macro that expands to an expression that extracts the value from the `value_set_accessor`.

CURRENT_INDEX — Equation macro

Arguments:

`index_set_h IndexSet`	Handle to an index set.

Returns:

A `index_t`.

Description:

Returns the current (numeric) index of the given index set. This is a macro that expands to an expression that either extracts the value from the `value_set_accessor` or registers a dependency of the surrounding `Equation` on this index set, depending on the contex of the evaluation.

FOREACH_INPUT — Equation macro

Arguments:

`index_set_h IndexSet`	Handle to a branched index set.
`Def`	The code to put inside of the for loop.

Description:

Expands to something that looks like

```
for(<every index in that is a branch input to CURRENT_INDEX(IndexSet)>)
{
        Def
}
```

Inside `Def` you can refer to the index of the iterated branch by `*Input`. Note that `FOREACH_INPUT` calls `CURRENT_INDEX(IndexSet)`, and thus registers an index set dependency.

INPUT_COUNT	Equation macro
Arguments:	
`index_set_h IndexSet`	Handle to a branched index set.
Returns:	
A `size_t`.	
Description:	
Returns the amount of branch input indexes to the current index in the index set. This is a macro that expands to an expression that either extracts the value from the `value_set_accessor` or registers a dependency of the surrounding `Equation` on this index set, depending on the contex of the evaluation.	

References

[1] M. N. Futter, M. A. Erlandsson, D. Butterfield, P. G. Whitehead, S. K. Oni, and A. J. Wade. PERSiST: a flexible rainfall-runoff modelling toolkit for use with the INCA family of models. *Hydrol. Earth Syst. Sci.*, 18:855–873, 2014.

[2] L. A. Jackson-Blake, J. E. Sample, A. J. Wade, R. C. Helliwell, and R. A. Skeffington. Are our dynamic water quality models too complex? a comparison of a new parsimonious phosphorous model, SimplyP, and INCA-P. *Water Resources Research*, 53:5382–5399, 2017.