Documentation for INCABuilder model builders - documentation is in development

Magnus Dahler Norling

February 22, 2019

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.

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. The different model entities are presented in Table 2.1

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 model. 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 for 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 for each time it is evaluated. There are several types of equations. The main two are discrete timestep equations and ordinary differential (ODE) equations.
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 the specific setup of a model. This includes all the indexes of each index set, as well as all the specific values of the parameters and input series. It will also contain the result series 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 provided in separate documentation.

3 Model structure

3.1 Index sets and indexes

Index sets is one of the fundamental concepts in the INCABuilder framework. Examples of index sets can 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.

> **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". If one of the inputs also index over "Reaches", then "Snow fall" will be evaluated once per reach-landscape unit pair, and so on.

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 model do however require a fixed set of particular indexes for some of its index sets. For instance INCA-N only works with three soil boxes, while INCA-C works with four.

There are two types of index sets

 i Basic index sets. These have no additional structure.

 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, which are 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 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
```

Here the parameter indexes over "Landscape units" (signifying what matrix you are looking at) and "Soils" twice (signifying which row and column). The percolation matrix is used says 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 (though one can usually configure which indexes each index set has).

3.3 Input structure

An input is a timeseries (usually of measured data or preprocessed 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 the equations having to do with flow or volume of the reach are evaluated for each reach index. 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 would have been 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 sets of an equation is 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 functio 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

```cpp
#define INCA_PRINT_TIMING_INFO 1

#include "../inca.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";

        inca_model *Model = BeginModelDefinition();

        AddTestModel(Model);

        ReadInputDependenciesFromFile(Model, InputFile);

        EndModelDefinition(Model);

        PrintResultStrucuture(Model);

        inca_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

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

        auto System = RegisterParameterGroup(Model, "System");
        RegisterParameterUInt(Model, System, "Timesteps", Days, 100);
        RegisterParameterDate(Model, System, "Start date", "1980-1-1");

        auto X = RegisterInput(Model, "X");
        auto A = RegisterParameterDouble(Model, System, "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 debug 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.

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 INCAView-compatible model application early, or make a inca.py - compatible dll for the model in order to do plots in python. 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 INCABuilder github repository.

4.2   Entity registration, handles, and equation bodies

The `inca_model` object contains lists of various model entities (see Table 2.1 for an explanaition) and information about these. It is the model developers 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 will be given later.

Each registration function returns a handle. In Example **??**, 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 reference 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 INCABuilder

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

Symbol	Type	Units	Description	Default	Min	Max
$D_{snow,0}$	parameter	mm	Initial snow depth	0.0	0.0	10000.0
f_{DDSM}	parameter	$mm\,d^{-1}\,{}^{\circ}C^{-1}$	Degree-day factor for snowmelt	2.74	1.6	6
$Pptn$	input timeseries	$mm\,d^{-1}$	Precipitation			
T_{air}	input timeseries	$^{\circ}C$	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 are given by the following equations

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

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

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

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

$$P = P_{rain} + P_{melt}$$

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 read from a parameter file before one runs the model, and the model 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 that equation for 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 `PrintModelResultStructure` 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 order of "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 reach, we would have got 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 has to do with how it registers what equations depend on what other entities. Right before the main model run, 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.

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 std::exp (the exponential function).

iv Always returning a value of type double (if you do not do this, you will get an 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 rule 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".

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 group 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 and y is the predator population. This can be implemented as follows

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

```
auto LVSolver = RegisterSolver(Model, "Lotka-Volterra␣solver", 0.1, IncaDascru);
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); //PredatirBaseGrowthRate 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 few 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 complicated computation can be reused by many ODE equations.

 ii Instead of returning the value of the 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.

For each time the solver evaluates its batch (i.e. for many sub-timesteps 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.

> **Note 4.** 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 4.6.

Which integration methods are available is subject to change. It is better that you look for them yourself in the source code, such as in `inca_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.

#TODO: Example

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 an unsigned integer between 0 and $N - 1$, where N is the number of indexes in the index set. Inside `EQUATION` bodies, indexes are for performance reasons always referred to by their number (i.e. the order they were declared) instead of their string name. 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 unsigned integer which tells you the number of indexes in the index set corresponding to the provided `IndexSetHandle`.

iii `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.

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

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 19. 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, RESULT(WaterDepth3));
        }
        return percolationOut;
)
```

In Example 19 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 to be 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 `double mpi = PARAMETER(Infiltration, OtherSoil)`. We are looking up the value of the parameter `Infiltration`, but instead of looking up the value for the current box 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 depends on `SoilBoxes` directly.

The parameter can have other index set dependencies, i.e. its parameter group can be a member of another parameter group which itself has an index set dependency. The index of this higher level index set is inferred to be the current index of the execution state. Moreover, this call will make register a dependency for `PercolationOut` on the higher level index set(s) that were automatically inferred, but not on `SoilBoxes`. However, `PercolationOut` does get a dependency on `SoilBoxes` any way from the call to `CURRENT_INDEX(SoilBoxes)`.

Example 20. 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 be evaluated for each index in "Reaches".

You can also provide more than one explicit index.

Example 21. 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 be evaluated for each index in "Reaches".

Example 21 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 22. Explicitly indexing a higher index only.

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

Note 5. The explicit index you provide always has to be an `index_t`, which is just a number, and so is indistinguishable between different index sets. This means that the framework will not be able to warn you if you use an index you got from a call to `CURRENT_INDEX(SoilBoxes)` when explicitly indexing a parameter that only depends on `LandscapeUnits`. Errors like these could cause undefined behaviour that is hard to track down, and so you should always be careful with explicit indexing.

Explicit indexing of `INPUTs` 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). Another difference is that an explicitly indexed `RESULT` or `LAST_RESULT` lookup will not register any index set dependencies whatsoever. The reason for this is that the order of the index set dependencies of an equation are determined after the dependency registration pass. So it is unable to determine during the dependency registration pass exactly which index set was referred to. So far we have not found a reliable way to fix this in the framework. If you have an equation that did not pick up the right dependency because of an access like this, you can use the following trick.

Example 23. Calling `CURRENT_INDEX` to force an index set dependency.

```
EQUATION(Model, OrganicLayerToMineralLayerFraction,
        CURRENT_INDEX(Reach); CURRENT_INDEX(LandscapeUnits); //NOTE: This forces the
            equation to depend on these two index sets.
        return RESULT(PercolationInput, MineralLayer) / RESULT(WaterDepth,
            OrganicLayer); //NOTE: We had to do that because we don't pick up any
            index set dependencies when we do explicit indexing of results.
)
```

Note 6. You should generally try to avoid explicit accesses of `RESULTs` inside equations that are in a solver batch. #TODO: Explain why. Explicit accesses of parameters are ok, but see the notes on performance under Section 4.8.

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.

#TODO: Write the section

> **Note 7.** 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 `inca_model.cpp`.

5 Model building API

6 Application building API

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.